Universidad Carlos III de Madrid
Escuela Politécnica Superior
Ingeniería en Informática

Grado Ingeniería Informática
Trabajo fin de grado

UN AGENTE EN JUEGO DIPLOMACY

Autor
Santiago Núñez Pulgar
Director
Javier Carbó Rubiera

Agradecimientos

Quiero dar las gracias a mi tutor Javier Carbó Rubiera por darme la oportunidad de desarrollar este proyecto.

A mi familia, por hacer esto posible.

Índice general

Índice de figuras

Índice de tablas

1. Introducción

1.1. Motivación

Diplomacy es un juego de estrategia por turnos de índole militar. El juego permite a siete jugadores controlar un país o potencia mundial, con el objetivo de conquistar Europa en el siglo XX, mediante la ocupación de 18 depósitos de suministros de los 34 disponibles a los largo del mapa europeo.

Diplomacy es un juego de suma cero, en teoría de juegos se considera un juego de suma cero donde la ganancia o pérdida de un jugador se equilibra con exactitud con las pérdidas o ganancias de los otros jugadores. En Diplomacy, sólo un jugador puede ganar, y si gana él, los demás pierden. Diplomacy no se puede considerar un juego cooperativo, ya que todas las decisiones de una potencia son independientes, una potencia no puede obligar a otra a cometer una acción. Como otros juegos de mesa, Diplomacy tiene un gran interés en el ámbito de la investigación computacional debido a su gran espacio de búsqueda de soluciones durante el turno de un jugador. Además de tener un comportamiento no determinista, ya que las órdenes dadas a una unidad pueden verse afectadas por otros jugadores y no llevarse a cabo.

La negociación entre jugadores es un aliciente para un entorno de agentes, donde la confianza entre ellos juega un papel importante en las negociaciones, y dado el gran espacio de soluciones supone un reto en el desarrollo de un agente.

1.2. Objetivo

Teniendo en cuenta que Diplomacy es un juego basado principalmente en negociaciones, el trabajo se basará en crear un jugador artificial que observando el entorno, determine las propuestas más favorables para él tras una negociación y proceda con las acciones que más le acerquen a su objetivo de ganar la partida.

El principal objetivo de este trabajo es desarrollar un agente con el framework DipGame [1], que tenga una arquitectura BDI (*Belief–desire–intention*) [2] y que pueda negociar con otros agentes en el nivel 1 del lenguaje especificado por DipGame.

1.3. Visión general del documento

El trabajo está dividido en ocho capítulos:

El primer capítulo representa la introducción del trabajo, explicando la motivación y objetivo del trabajo.

El segundo capítulo explica el estado de la cuestión: los conceptos de agente software, topología de agentes, sistemas multi-agente y el modelo de agente que se usará en el trabajo.

El tercer capítulo detalla una visión general del juego Diplomacy; explicando su interés en el ámbito computacional, reglas y diferentes componentes del juego.

El cuarto capítulo explica el estado del arte en cuanto a agentes usados en Diplomacy y explica el framework a utilizar, DipGame.

El quinto capítulo supone el detalle de la implementación del agente, explicando su arquitectura y metodología para conseguir el objetivo de ganar la partida.

El sexto capítulo representa las pruebas realizadas con el agente implementado y sus resultados.

El séptimo capítulo representa las conclusiones alcanzadas sobre el trabajo, las posibles mejoras y trabajos futuros sobre él y los problemas encontrados en la realización del trabajo.

2. Estado de la cuestión

2.1. Agente software

Existen múltiples definiciones sobre qué es un agente software, aquí se adaptará la definición establecida por Michael Wooldridge y Nick Jennings [2]:

Un agente software es una entidad que percibe el entorno en el que se encuentra a través de unos sensores, y actúa sobre dicho entorno a través de actuadores. Un agente debe ser capaz de resolver problemas que se encuentra, de forma autónoma y exhibiendo un comportamiento flexible.

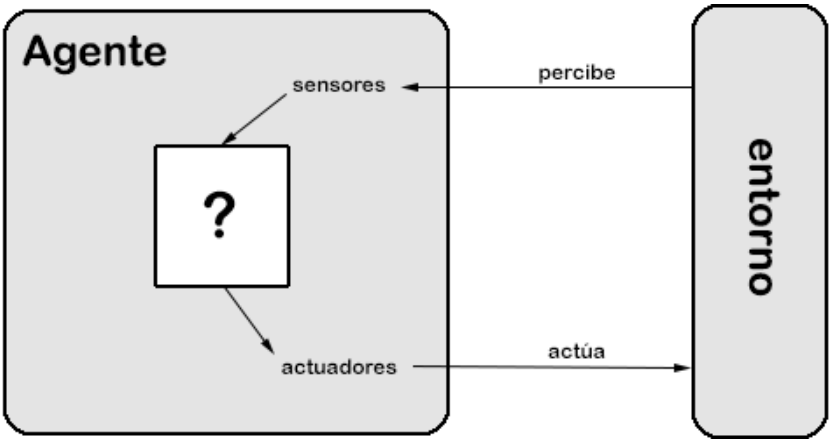


Figura 1. Un agente interactúa con el entorno a través de sensores y actuadores.

En la Figura 1 observamos de forma abstracta el comportamiento de un agente en el entorno. El agente deberá decidir a partir de sus percepciones que salida generar para actuar sobre el entorno, y sin un control total sobre él, influir en el mismo para conseguir sus objetivos.

Según Stuart Russell y Peter Norvig, el entorno de un agente puede ser [3]:

- Accesible o inaccesible
    - El agente puede recuperar toda la información posible y actualizada del entorno y su estado.

- Determinista o no determinista
    - En un entorno determinista, una acción tendrá siempre un resultado fijo e invariable. No habrá una incertidumbre sobre el estado del entorno tras la acción.
- Estático o dinámico
    - Un entorno estático solo cambia cuando el agente realiza acciones sobre él, sin embargo, un entorno dinámico cambia constantemente sin la necesidad de una acción del agente.
- Discreto o continuo
    - Un entorno discreto dispone de un número finito de acciones que el agente puede realizar sobre él, por lo tanto dispone de un número finito de estados posibles, frente a la infinidad de estados en un entorno continuo.

2.1.1. Características de un agente

Las características más comunes de un agente son:

- Autonomía
    - Un agente debe tomar decisiones sin la intervención de terceras partes.
- Persistencia
    - El agente debe ser un proceso continuamente en ejecución. La percepción del entorno debe ser persistente y no bajo demanda.
- Reactividad
    - Tiene que ser capaz de reaccionar a los estímulos recibidos del entorno y actuar acordemente.
- Proactividad
    - Debe ser capaz modificar sus objetivos, tomando la iniciativa

Otras características deseables para un agente son:

- Racionalidad
    - El agente tiene un conocimiento de su entorno, posee unas reglas que le permiten alcanzar unos objetivos a partir del conocimiento que posee.
- Aprendizaje
    - Puede actualizar su conocimiento y cambiar su comportamiento en base a experiencias pasadas.
- Sociabilidad
    - La capacidad de establecer comunicación con otros agentes.
- Movilidad
    - Puede ser capaz de cambiar de plataforma de otra.

2.2. Clasificación de agentes

Existen múltiples clasificaciones de los agentes, las más comunes lo hacen en base a sus características. Una de las clasificaciones más aceptadas es la propuesta por Hyacinth Nwana [4]:

- Por su movilidad:
    - Estáticos
        - No son capaces de cambiar de plataforma.
    - Móviles
        - Pueden migrar a otra máquina para ejecutar las tareas encomendadas.
- Por su toma de decisiones:
    - Reactivo
        - Responden a los estímulos del entorno de manera directa y preconcebida.
    - Deliberativo
        - Tienen un estado simbólico interno y un modelo de razonamiento que les permiten planificar las decisiones, además de negociar y coordinar objetivos con otros agentes.
- Por su comportamiento:
    - Autónomo
    - Capaz de aprender
    - Cooperativo

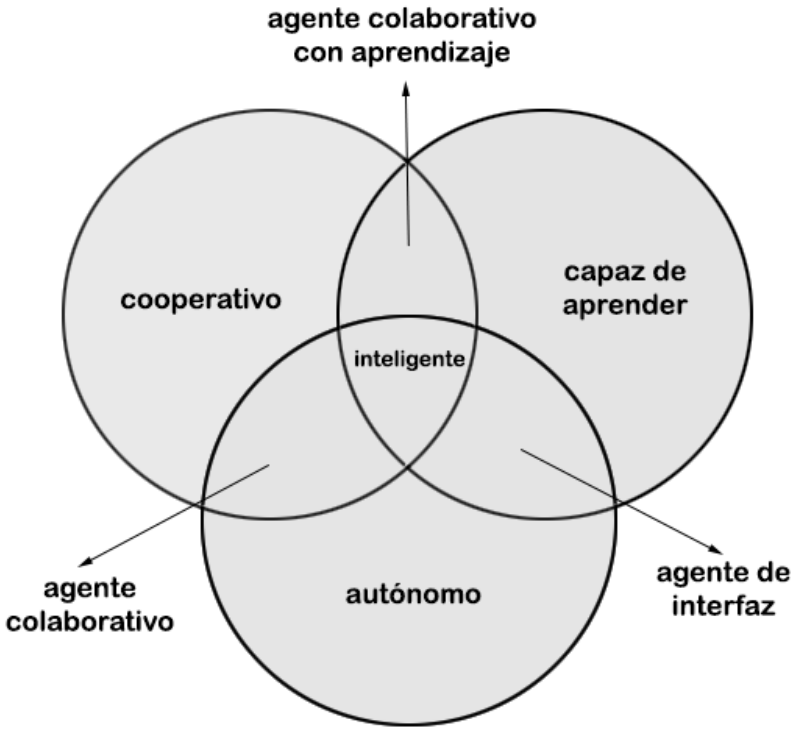


Figura 2. Clasificación de agentes por Hyacinth Nwana.

- Agente colaborativo con aprendizaje
    - Un agente colaborativo con aprendizaje es capaz de encontrar patrones en el entorno y con la ayuda de otros agentes, identificar dichos patrones o nuevos patrones en sucesivos cambios del entorno.
- Agente colaborativo
    - Los agentes colaborativos son una parte esencial de los sistemas multi-agente. Usan la colaboración con otros agentes y su propia autonomía como el medio para solucionar el problema.
- Agente de interfaz
    - Los agentes de interfaz son una especialización de agentes orientados a la ayuda del usuario. Usan su capacidad de aprendizaje para mejorar la experiencia y ayuda que necesita el usuario de ellos.

La clasificación de los agentes no tiene por qué ser excluyente, un agente puede poseer varias características que lo conviertan en un agente híbrido. Un agente colaborativo pone más énfasis en la autonomía y cooperación que en el aprendizaje.

Idealmente un agente debería poner en mismo énfasis en todas las áreas, lo que se denomina un agente inteligente, pero en la realidad supone un reto difícil de alcanzar.

Walter Brennan et al. proponen otra clasificación [5]:

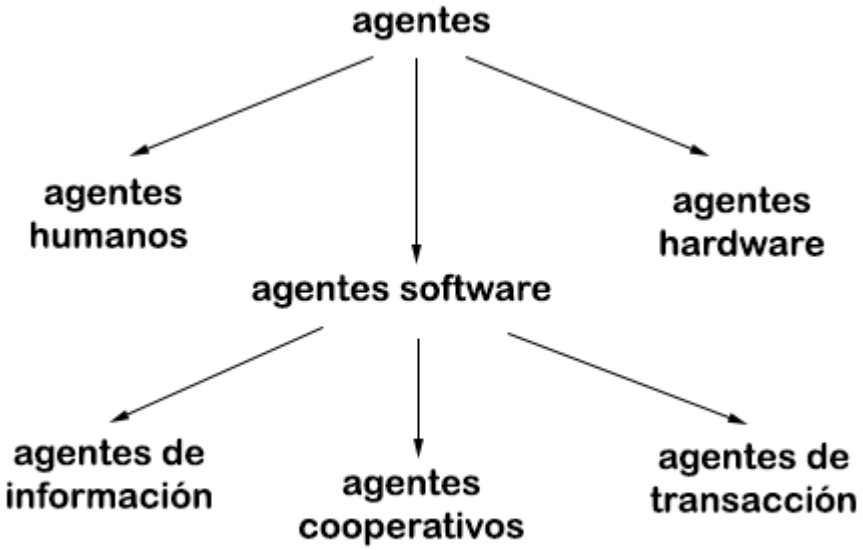


Figura 3. Clasificación de agentes por Brennan et al.

Como se puede observar, las clasificaciones más comunes se basan en características propias de los agentes.

2.3. Arquitectura Belief–desire–intention

Un agente con una arquitectura BDI utiliza los conceptos de creencias (*beliefs*), deseos (*desires*) e intenciones (*intentions*) para para decidir la acción a ejecutar en un entorno dado. En principio surgió como un modelo filosófico propuesto por Michael Bratman [6] para explicar el comportamiento humano a través de los tres conceptos mencionados: creencias, deseos e intenciones. La implementación en agentes fue propuesta por Anand Rao y Michael Georgeff en *BDI Agents: From Theory to Practice* [7], introduciendo la teoría formal y el concepto de interpretador abstracto.

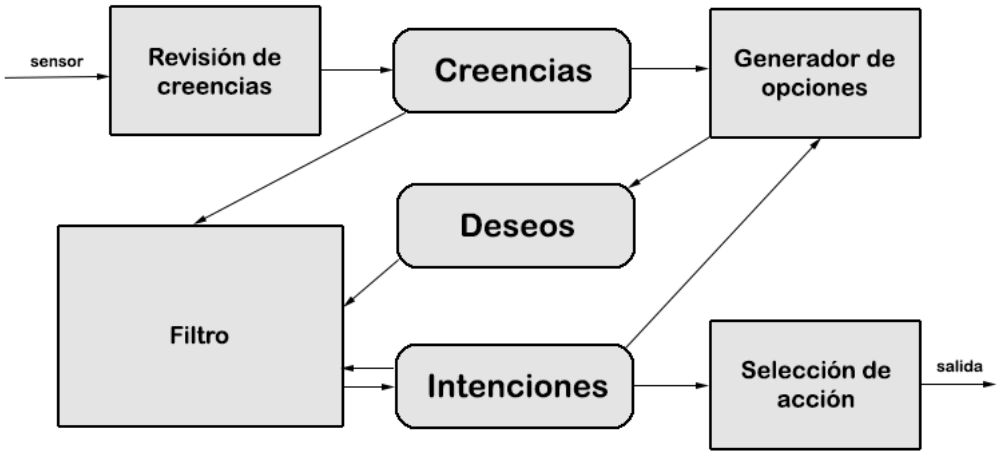


Figura 4. Arquitectura de un agente BDI.

Las creencias del agente son características del entorno que percibe el agente y se actualizan según considere el agente mediante una función de revisión de creencias. Se consideran el componente informativo del sistema.

Los deseos del agente son la información relativa a los objetivos que quiere alcanzar el agente. Se generan mediante una función generadora de opciones a partir de sus creencias e intenciones. Representan el componente motivacional del sistema.

Las intenciones del agente son la representación de los planes de acciones seleccionados por el agente para ejecutar mediante una función de filtrado. Representan el componente deliberativo del sistema.

Un agente BDI percibe hechos del entorno que considera como creencias, a partir de esas creencias. A partir de dichas creencias y las propias intenciones actuales del agente, el agente genera una serie de opciones (deseos). Con las opciones disponibles, el agente deliberará nuevas intenciones en el proceso de filtrado. Una vez obtenidas las intenciones actuales del agente, sólo le queda determinar la acción a ejecutar de las opciones disponibles.

- Inicializar_estado()
- Repetir
    - Intención = INTENCIONES.Siguiente(CREENCIAS)
    - Ejecutar(intención)
    - percepción = PERCEPCIONES.Percibir()
    - CREENCIAS.Actualizar(percepción)
    - INTENCIONES.Actualizar(CREENCIAS)
    - DESEOS.Actualizar(INTENCIONES)
    - INTENCIONES.Nuevas(DESEOS)
    - INTENCIONES.Nuevas(DESEOS)
- Fin Repetir

Tabla 1. Algoritmo de un agente deliberativo BDI.

2.4. Sistema multi-agente

Se puede definir un sistema multi-agente (SMA) como una organización de agentes, que formando un sistema, interaccionan entre sí.

Jacques Ferber considera un sistema multi-agente cuando cumple con los siguientes requisitos [8]:

- Contiene un entorno en el que se sitúan todas las entidades.
- Contiene una serie de objetos, entidades pasivas, que pueden ser percibidos por los agentes.
- Contiene un conjunto de agentes, entidades activas, que se encargan de actuar sobre los objetos del entorno.
- Relaciones entre todas las entidades.
- Conjunto de operaciones que pueden ser realizadas por los agentes sobre los objetos y permiten el cambio del entorno.

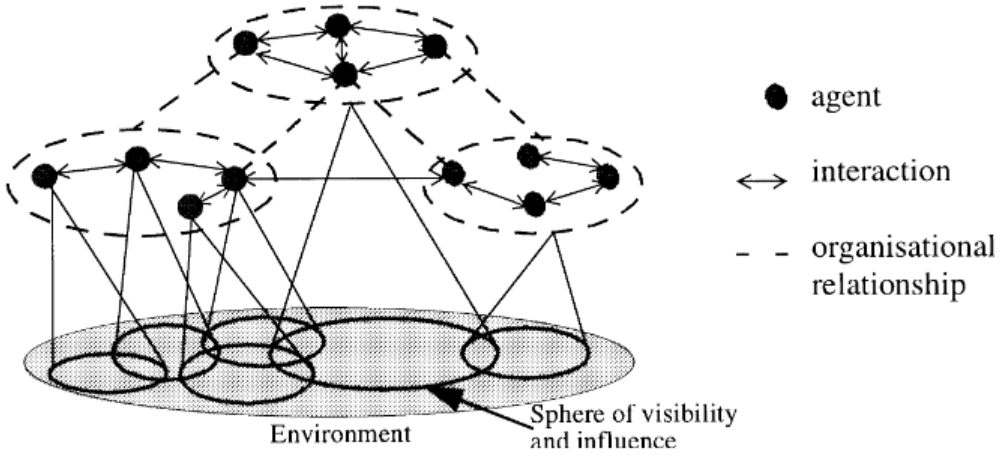


Figura 5. Estructura de un sistema multi-agente [9].

Un sistema multi-agente permite la resolución que problemas que un agente de forma individual sería incapaz de resolver.

Los agentes de un sistema pueden interactuar entre sí dependiendo de la tarea a realizar:

- Objetivos compatibles o incompatibles
- Recursos ofrecidos por el entorno
- Habilidades para realizar la tarea

O dependiendo de la relación entre los agentes:

- Convivencia
    - Los agentes conviven en el entorno.

- Conocimiento
    - Un agente puede conocer la existencia de otro e interactuar con él.

- Comunicación
    - Un agente puede comunicarse con otro.

- Subordinación
    - Un agente puede realizar una tarea que le pide otro por medio de negociación (subordinación dinámica) u obligado (subordinación estática).

- Información
    - Un agente puede informarse del conocimiento de otro, por lo que sus creencias dependerán de otros agentes.

De esta manera se formulan varios tipos de interacción entre los agentes:

- Independencia
    - Los objetivos de los agentes son compatibles, el entorno dispone de recursos suficiente para todos ellos y todos tienen la habilidad necesaria para conseguir sus objetivos.

- Colaboración simple
    - Los objetivos son compatibles y los recursos suficientes, pero la habilidad de los agentes no les permite solucionar la tarea independientemente. Por ello los agentes aportan sus habilidades para el bien común.

- Obstrucción
    - Los agentes tienen objetivos compatibles y la habilidad necesaria para resolver la tarea, pero los recursos del entorno no son suficientes para todos. Esto crea una competición entre ellos, por lo que requieren de una coordinación.

- Colaboración coordinada
    - Con objetivos compatibles pero con unos recursos insuficientes para todos, además de una falta de habilidad para resolver la tarea independientemente, los agentes deberán coordinarse para conseguir los objetivos.

- Competición individual

    - Los objetivos son incompatibles, los recursos suficientes y cada uno tiene la habilidad requerida para completar la tarea. Esto supone que los agentes actúan de forma individual para completar sus objetivos a costa de los demás.

- Competición por equipos
    - Los objetivos son incompatibles y los recursos suficientes para todos, pero la habilidad de cada agente no es la necesario para que cumplan la tarea. Los agentes deberán formar equipos para conseguir los objetivos de manera coordinada.

- Conflicto individual
    - Los objetivos son incompatibles y los recursos insuficientes. Sin embargo los agentes disponen de la habilidad necesaria para conseguir el objetivo. Esto supone que los agentes competirán por los recursos del entorno.

- Conflicto colectivo
    - Los objetivos son incompatibles, los recursos insuficientes y las habilidades insuficientes. Los agentes deben formar coaliciones según objetivos para luchar por los recursos.

Estas interacciones obligan a los agentes a tener una capacidad de negociación.

2.5. Negociación entre agentes

La distribución de tareas entre los agentes requiere de una colaboración, coordinación y negociación entre ellos.

La colaboración entre los agentes supone la consecución de unos objetivos comunes por medio del trabajo aportado por todos los agentes pero de forma individual.

A diferencia de la colaboración, la coordinación entre los agentes supone la distribución de la carga de trabajo entre los distintos agentes para la consecución de los objetivos.

La negociación entre los agentes permite la resolución de conflictos, definición de tareas y métodos de resolución de problemas. Es la discusión entre dos partes para alcanzar un mutuo acuerdo.

Según Michael Wooldridge, la negociación entre agentes requiere [10]:

- Una ontología que represente el espacio de posibles mensajes entre los agentes.
- Un protocolo que defina las propuestas legales que pueden hacer los agentes.
- Una estrategia de negociación, una por agente, que defina las propuestas que un agente hará. Esta estrategia suele ser privada del conocimiento de otros agentes.
- Una regla que determine el estado de un trato y el tipo de trato que es.

En una negociación los agentes se propondrán propuestas por rondas. Dichas propuestas estarán definidas por la estrategia del agente, formadas con la ontología de la negociación y de forma correcta según el protocolo. Cuando se alcanza un acuerdo, la negociación termina.

3. Diplomacy

Diplomacy es un juego militar de estrategia creado por Allan B. Callamer en 1954 y publicado en 1959. Mediante turnos, siete potencias del mundo luchan por conquistar Europa durante el siglo XX, justo después de la primera guerra mundial. Siendo al principio un juego de tablero tradicional, su popularidad llegó a tal punto de jugar partidas mediante cartas por correo, donde los turnos podrían llegar a durar semanas [11]. La primera versión para PC fue publicada en 1984 por la compañía *Avalon Hill*. Hoy en día los derechos pertenecen a *Wizards Of the Coast*.

3.1. Interés en el campo de la computación

El juego despierta un gran interés en el ámbito de la informática dada su complejidad. El ajedrez, otro juego muy interesante, tiene un factor de ramificación medio de 35 y una profundidad media de 80 [12], es decir, un jugador tiene de media 35 movimientos legales diferentes en su turno. Al inicio de una partida de ajedrez, el factor de ramificación es de 20 (2 movimientos por cada uno de los 8 peones más 2 movimientos por cada uno de los 2 caballos). Al ser un juego secuencial, hay un total de 20^2 aperturas diferentes (20 movimientos del primer jugador por 20 posibles movimientos del segundo). En cambio en el juego Diplomacy, en el inicio de una partida se tienen 22 unidades en el mapa y cada una tiene a su disposición 10 movimientos diferentes, al ser turnos simultáneos se ha calculado en 4430690040914420 las combinaciones posibles de turnos [13]. El gran tamaño que puede alcanzar el árbol de decisión en el juego propicia una investigación alternativa a los típicos algoritmos de búsqueda de soluciones. Por ejemplo mediante negociaciones entre los jugadores, un trabajo adecuado para agentes.

3.2. Reglas básicas del juego

En esta sección se explicarán brevemente las reglas de una versión estándar de Diplomacy que salvo alguna diferencia son las usadas por el banco de pruebas DipGame.

El juego se centra en un mapa de Europa, en el que siete potencias luchan por la conquista de los depósitos de suministros. Los jugadores son asignados aleatoriamente a las potencias: Alemania, Austria, Francia, Inglaterra, Italia, Rusia y Turquía. Cada potencia comienza con tres unidades excepto Rusia que comienza con cuatro. Dichas unidades se situarán en los depósitos de suministros controlados inicialmente por las potencias, es decir cada potencia comienza con tres depósitos de suministros excepto Rusia que tendrá cuatro. El número total de depósitos de suministros es de 34, de los cuales 22 son controlados inicialmente por las potencias. El objetivo del juego es controlar al menos 18 depósitos de los 34. La primera potencia en lograrlo será la ganadora y el resto serán ordenados por número de depósitos obtenidos el año en el que fueron eliminadas del juego.

3.2.1. Mapa

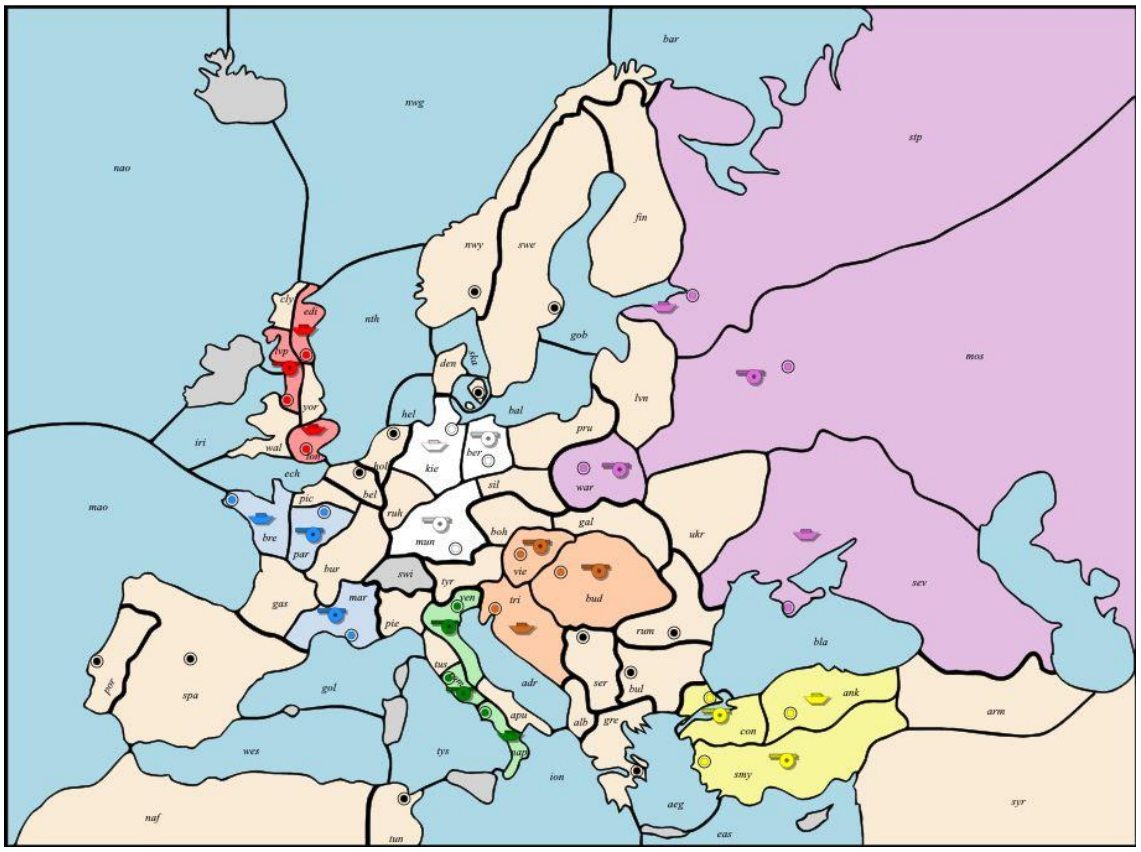


Figura 6. Mapa de una partida de Diplomacy en DipGame.

El mapa del juego es Europa dividida en 74 regiones. En la Figura 6 observamos el estado inicial de una partida en DipGame. Los círculos del mapa son los depósitos de suministros y los cañones o flotas son las unidades de cada potencia. El color beige indica regiones sin ocupar y las líneas separadoras indican la vecindad de cada región.

Las regiones pueden ser de tres tipos: regiones interiores, regiones de costa y regiones marítimas. Las regiones interiores son regiones de tierra que no tienen vecindad con el mar, las regiones de costa son regiones de tierra que tienen a una región marítima en algún vecino y una región marítima son regiones formadas por mar.

3.2.2. Fases de un turno

El juego comienza en el año 1901 y simula el paso de los años mediante turnos. Cada turno correspondería a una de las cinco fases en las que se divide el año:

- Primavera (*Spring*)
  - En esta fase las potencias podrán comunicarse entre ellas para pactar alianzas u otro tipo de comunicación y se dictan las órdenes a las unidades.

- Verano (*Summer*)
  - Se ejecutan las órdenes realizadas y se resuelven los conflictos al ser turnos simultáneos. También se disuelven las unidades ya sea porque el jugador lo quiera o por alguna regla del juego (por ejemplo si dos unidades deben retirarse a la misma provincia se disuelven ambas).

- Declive (*Fall*)
  - Fase de diplomacia y resolución de órdenes.

- Otoño (*Autumn*)
  - Resolución de conflictos y disolución de unidades.

- Invierno (Winter)
  - En esta fase cada jugador debe ajustar su número de unidades al número de depósitos que controle, es decir podrá crear unidades en depósitos que no estén ocupados por alguna unidad y tendrá que disolver unidades hasta que el número de ellas sea igual que el número de depósitos.

3.2.3. Unidades

Existen dos tipos de unidades en el juego: ejércitos y flotas, representadas por un cañón y un barco respectivamente. Los ejércitos pueden ocupar regiones de interior y de costa, y los barcos pueden ocupar regiones de costa y marítimas. Todas las unidades tienen la misma fuerza de combate y sólo una puede ocupar una región a la vez. En determinados casos como en los de España, Bulgaria y San Petersburgo, en el que la región tiene más de una costa con distintas fronteras, las flotas deberán posicionarse sobre la costa en vez de la región interior, sin embargo, independientemente de la costa elegida la región se considerará ocupada. Todas las unidades tienen un movimiento de una región, es decir, sólo pueden moverse a un vecino de la región en las que están situadas.

Las potencias tienen asignadas un color: Alemania el blanco, Austria el naranja, Francia es el color azul, Inglaterra el rojo, Italia el verde, Turquía el amarillo y Rusia el violeta.

Potencia	Color
Alemania	Blanco
Austria	Naranja
Francia	Azul
Inglaterra	Rojo
Italia	Verde
Turquía	Amarillo
Rusia	Violeta

Tabla 2. Colores asignados a las potencias en DipGame.

3.2.3.1. Posiciones iniciales

Cada potencia comienza la partida con unos depósitos de suministros asignados, y una unidad por cada uno de ellos. Como se ha mencionado antes, todas las potencias empiezan con tres depósitos excepto Rusia que comienza con cuatro. Por lo tanto, al inicio de una partida doce depósitos de suministros no tienen dueño.

Potencia	**Provincia**
Alemania	Berlín
	Múnich
	Kiel (Flota)
Austria	Viena
	Budapest
	Trieste (Flota)
Francia	París
	Marsella
	Brent (Flota)
Inglaterra	Londres (Flota)
	Edimburgo (Flota)
	Liverpool
Italia	Roma
	Venecia
	Nápoles (Flota)
Rusia	Moscú
	Sebastopol (Flota)
	Varsovia
	San Petersburgo (Flota)
Turquía	Ankara (Flota)
	Constantinopla
	Esmirna

Tabla 3. Depósitos y unidades iniciales de cada potencia en Diplomacy.

3.2.4. Órdenes

Las órdenes son el mandato que el jugador envía a cada unidad para su cumplimiento. Cada unidad podrá obedecer como máximo una orden en la fase de resolución de órdenes. Las órdenes posibles son:

- Mantener la posición (*Hold*)
  - Una unidad que mantiene la posición no realiza ningún movimiento y permanece en la región que está.

- Mover (*Move*)
  - Está acción permite a una unidad mover a una región vecina de la región en la que se encuentra. Cuando una unidad mueve a una región desocupada, la ocupa inmediatamente, sin embargo, si está ocupada u otras unidades se movieron a la misma región de destino, se produce un enfrentamiento. Todas las unidades tienen el mismo poder de ataque, por lo que es necesario superar en poder de ataque a la otra unidad para ocupar la región. Esto se soluciona con la orden de apoyo (support).

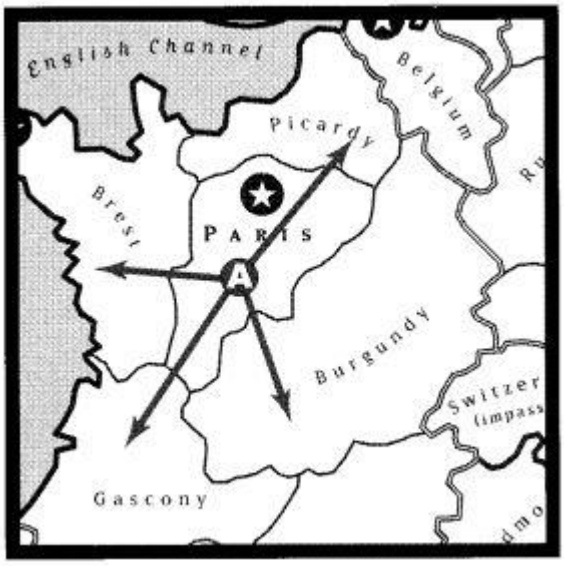


Figura 7. Posibles movimientos de una unidad terrestre en París. [14]

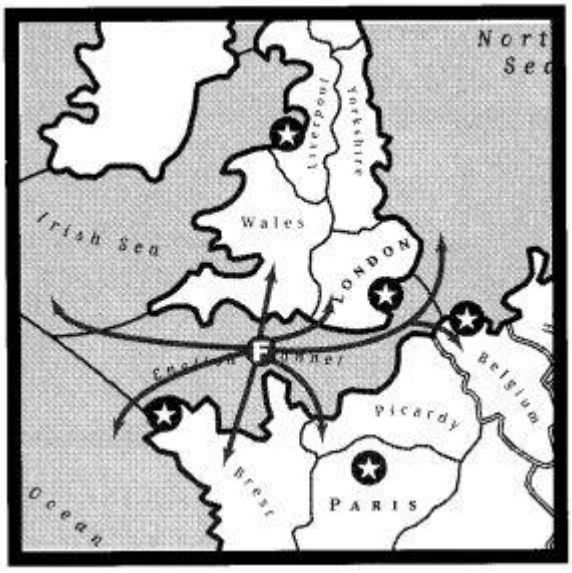


Figura 8. Posibles movimientos de una flota en el Canal de la Mancha. [14]

- Apoyo (*Support*)
  - Una unidad puede apoyar el movimiento o la manutención de la posición de otra. Apoyando a una unidad se incrementa el poder de ataque de esta en uno. El apoyo de una unidad a otra se puede obstaculizar con el ataque a la región donde se encuentra la unidad que quiere realizar el apoyo, impidiendo que su poder de ataque se sume a la unidad que pretendía apoyar. De esta manera se incrementa y reduce el poder de ataque que puede tener una unidad.

Figura 9. Una unidad en Gascuña apoya a la unidad en Marsella. [14]

- Retirada (*Retreat*)
    - Durante un enfrentamiento, una unidad puede salir derrotada y deberá mover a una región adyacente que no esté ocupada. En caso de que todas las regiones estén ocupadas, la unidad deberá desbandarse, y por lo tanto el jugador perderá la unidad.

- Construcción (*Build*)
    - En la fase de invierno el jugador podrá construir nuevas unidades hasta alcanzar el número de depósitos de suministros que controle.

- Convoy
    - Un convoy es el transporte de una unidad terrestre por una flota entre dos regiones a través de una región marítima. El framework DipGame no permite este tipo de orden.

- Desbandar (*Disband*)
    - Un jugador que tenga más unidades que depósitos de suministro o una unidad que no pueda retirarse a ninguna provincia adyacente tendrá que desbandarse y ser retirada de la partida.

- No construir (*Waive*)
    - En la fase de invierno un jugador puede decidir no construir una unidad a pesar de tener un depósito de suministro libre para hacerlo.

4. Agentes en Diplomacy

Dado el éxito que ha tenido el juego en el ámbito de la computación, se han diseñado plataformas que permiten la creación y prueba de jugadores artificiales mediante agentes. DAIDE (Diplomacy AI Development Environment) es la más conocida de estas plataformas y es la base del banco de pruebas DipGame, que será usado en este proyecto.

4.1. DAIDE

DAIDE es un entorno para el diseño de sistemas multi-agente en el juego Diplomacy [15]. Fue desarrollado como un hobby por un grupo de programadores para permitir que varios jugadores artificiales pudiesen competir entre sí. El entorno ofrece un protocolo de comunicación basado en TCP/IP con el que los clientes podrán comunicarse y un servidor al que los clientes se conectarán. Lo más importante que ofrece DAIDE es el lenguaje de comunicación que los agentes usarán entre ellos. El lenguaje ofrece hasta catorce niveles de comunicación (Tabla 4).

Nivel 0	Base
Nivel 10	Propuestas de alianza y paz
Nivel 20	Propuestas de ordenes
Nivel 30	Acuerdos
Nivel 40	Reparto de depósitos de suministros
Nivel 50	Acuerdos anidados
Nivel 60	Solicitudes y exigencias
Nivel 70	Solicitud de sugerencias
Nivel 80	Acusaciones
Nivel 90	Discusiones futuras
Nivel 100	Condicionales
Nivel 110	Marionetas y favores
Nivel 120	Reenvío de mensajes
Nivel 130	Explicaciones

Tabla 4. Niveles de lenguaje en DAIDE

Cada nivel añade un tipo de mensaje nuevo que el nivel anterior, por lo tanto, un nivel contiene los mensajes de los niveles anteriores [16]. La mayoría de los agentes implementados funcionan en nivel 0 y el máximo nivel implementado es 30 [17] en el cliente *Albert* [18].

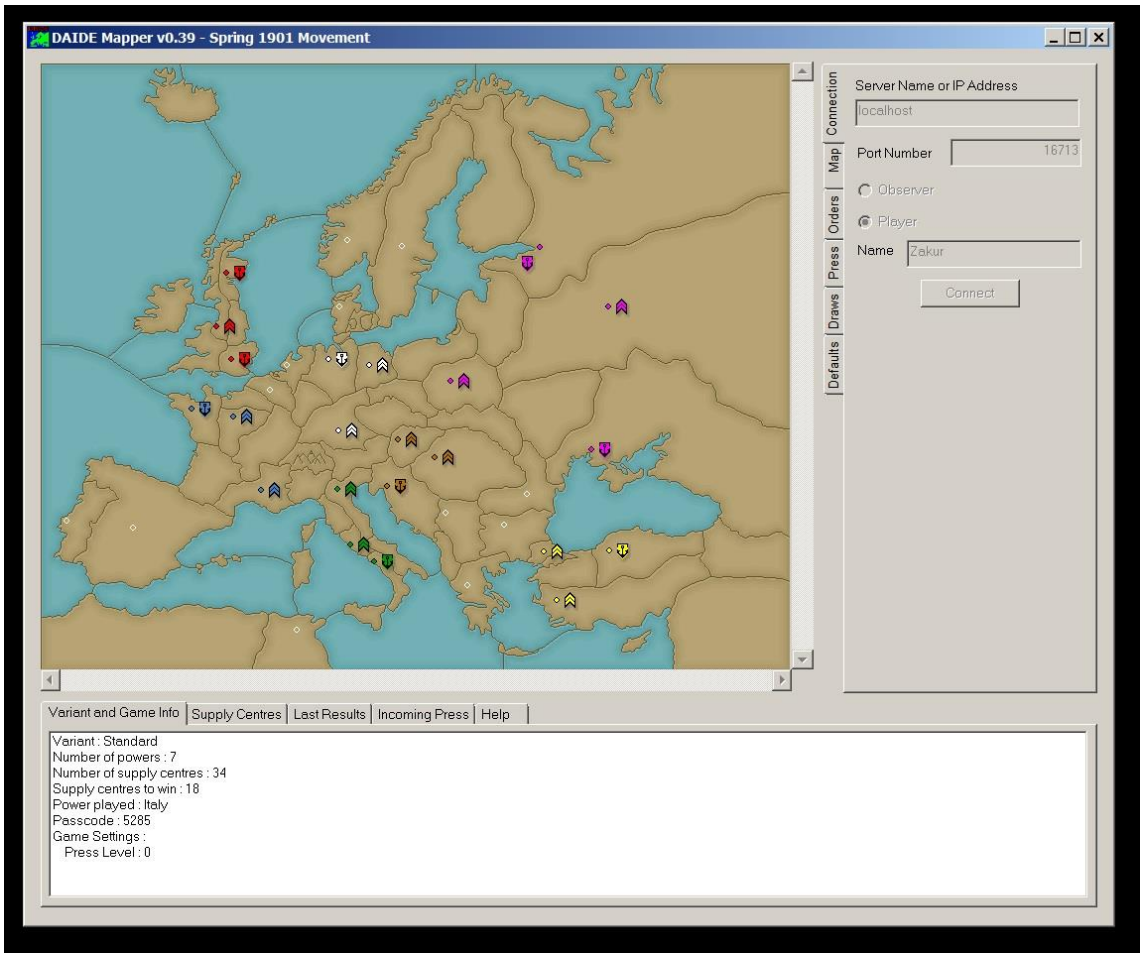


Figura 10. *Mapper* de DAIDE.

4.2. DipGame

DipGame es un entorno y banco de pruebas desarrollado por Angela Fabregues y Carles Sierra [1] en el Instituto de Investigación en Inteligencia Artificial (IIIA) de Barcelona, el cual pertenece al Consejo Superior de Investigaciones Científicas (CSIC).

Está basado en DAIDE y facilita el desarrollo de jugadores artificiales al estar implementado en Java, ofreciendo un desarrollo multiplataforma. DipGame usa su propio nivel de lenguaje, llamado L, sustituyendo al lenguaje ofrecido por DAIDE [Tabla 4].

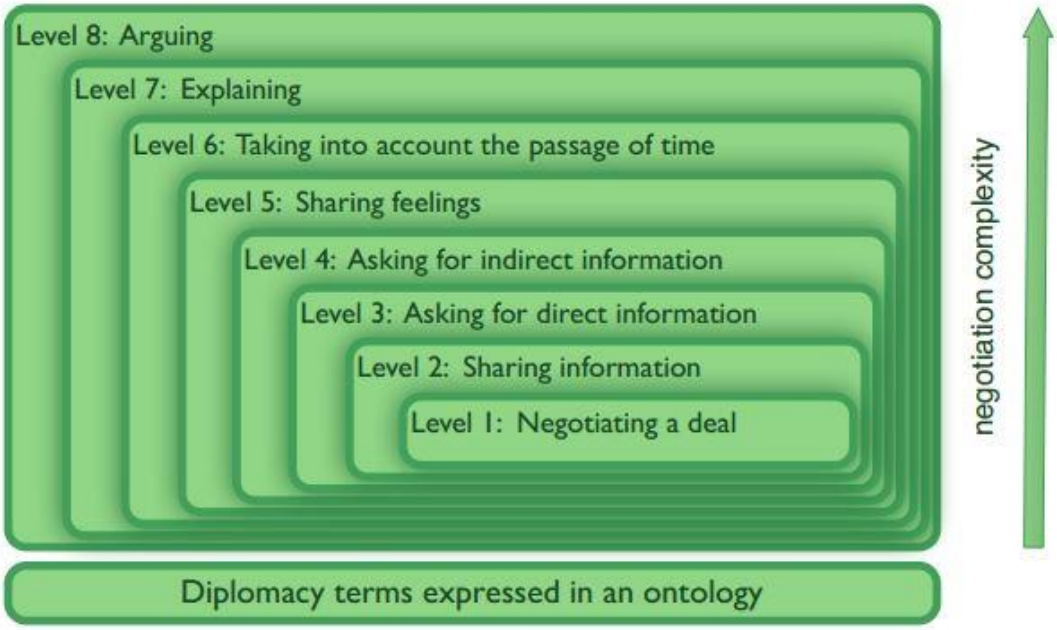


Figura 11. Niveles del lenguaje L en DipGame

L es un lenguaje genérico que define los actos que los agentes pueden usar para comunicarse entre ellos [19].

agent ::= power

action ::= order

predicate ::= offer

time ::= { phase, year }

Level 1: Negotiating a deal

L_1 ::= propose(α, β, $deal_1$) | accept(α, β, $deal_1$) | reject(α, β, $deal_1$) | withdraw(α, β)

$deal_1$ ::= Commit(α, β, ϕ)+ | Agree(β, ϕ)

ϕ ::= predicate | Do(action) | ϕ ∧ ϕ | ¬ϕ

β ::= α+

α ::= agent

Level 2: Sharing information

L_2 ::= L_1| inform(α, β, $info_2$)

$info_2$ ::= $deal_1$ | Obs(α, β, ϕ) | Belief(α, ϕ) | Desire(α, ϕ) | $info_2$ ∧ $info_2$ | ¬$info_2$

Level 3: Asking for direct information

L_3 ::= L_2| inform(α, β, $info_3$) | query(α, β, $info_3$) | answer(α, β, $info_3$)

$info_3$::= $info_2$ | Unknown(α, $info_3$) | $info_3$∧ $info_3$| ¬$info_3$

Level 4: Asking for indirect information

L_4 ::= L_3| inform(α, β, $info_4$) | query(α, β, $info_4$) | answer(α, β, $info_4$)

| inform (α, β, L_4) | query(α, β, L_4) | answer(α, β, L_4)

$info_4$::= $info_3$ | Unknown(α, $info_4$) | Unknown(α, L_4) | $info_4$∧ $info_4$ | ¬$info_4$

Level 5: Sharing feelings

L_5 ::= L_4| inform(α, β, $info_5$) | query(α, β, $info_5$) | answer(α, β, $info_5$)

| inform (α, β, L_5) | query(α, β, L_5) | answer(α, β, L_5)

$info_5$ ::= $info_4$| Unknown(α, $info_5$) | Unknown(α, L_5) | Feel(α, feeling)

| $info_5 \wedge info_5$ | $\neg info_5$

feeling ::= VeryHappy | Happy | Sad | Angry

Level 6: Taking into account the passage of time

L_6 ::= L_5 | propose(α, β, $deal_6$, t) | accept(α, β, $info_6$, t) | reject(α, β, $deal_6$, t)

| withdraw (α, β, t) | inform (α, β, $info_6$, t) | query(α, β, $info_6$, t)

| answer (α, β, $info_6$, t) | inform(α, β, L_6, t) | query(α, β, L_6, t)

| answer (α, β, L_6, t)

$info_6$ ::= $info_5$ | $deal_6$ | Obs(α, β, ϕ6, t) | Belief(α, ϕ6, t) | Desire(α, ϕ6, t)

| Unknown (α, $info_6$, t) | Unknown(α, L_6, t) | Feel(α, feeling, t)

| $info_6 \wedge info_6$ | $\neg info_6$

$deal_6$ ::= $deal_5$ | Commit(α, β, ϕ6, t)+ | Agree(β, ϕ6, t)

Level 7: Explaining

L_7 ::= L_6 | inform(α, β, $info_7$, t) | query(α, β, $info_7$, t) | answer(α, β, $info_7$, t)

| inform (α, β, L_7 , t) | query(α, β, L_7 , t) | answer(α, β, L_7 , t)

$info_7$ ::= $info_6$| Unknown(α, $info_7$, t) | Unknown(α, L_7 , t) | Explain($info_7$, t)

| Explain (L_7 , t) | $info_7 \wedge info_7$ | $\neg info_7$

Level 8: Arguing

L_8 ::= L_7 | inform(α, β, $info_8$, t) | query(α, β, $info_8$, t) | answer(α, β, $info_8$, t)

| inform (α, β, L_8 , t) | query(α, β, L_8 , t) | answer(α, β, L_8 , t)

$info_8$ ::= $info_7$ | Unknown(α, $info_8$, t) | Unknown(α, L_8 , t) | Explain($info_8$, t)

| Explain (L_8 , t) | Attack ($info_7$,$info_7$) | Support ($info_7$, $info_7$)

| $info_8 \wedge info_8$ | $\neg info_8$

Figura 12. Sumario reducido de los niveles del lenguaje L en DipGame [19]

El primer nivel, “negociar un trato”, ofrece a un agente proponer a otro la realización de unas órdenes del juego o la formación de una alianza.

El segundo nivel, “compartir información”, ofrece a los agentes compartir información de la que tengan conocimiento como tratos previos, acciones observadas, deseos o creencias.

El tercer nivel, “preguntar por información directa”, permite a un agente requerir información a otro sobre su conocimiento y contestarle.

El cuarto nivel, “preguntar por información indirecta”, ofrece a un agente requerir información a otro agente sobre sus acciones con otros agentes.

El quinto nivel, “compartir sentimientos”, permite a un agente preguntar los sentimientos de otro respecto a alguna información.

El sexto nivel, “teniendo en cuenta el paso del tiempo”, permite a un agente comunicar predicados respecto al tiempo.

El séptimo nivel, “explicando”, permite a los agentes explicar hechos a otros agentes.

El octavo nivel, “discutiendo”, ofrece a los agentes la posibilidad de rebatir y apoyar argumentos de otros agentes.

4.2.1. Infraestructura

DipGame como framework ofrece varias clases con las que un desarrollador podrá diseñar e implementar un agente con la capacidad de jugar al juego Diplomacy y poder negociar con otros agentes del juego.

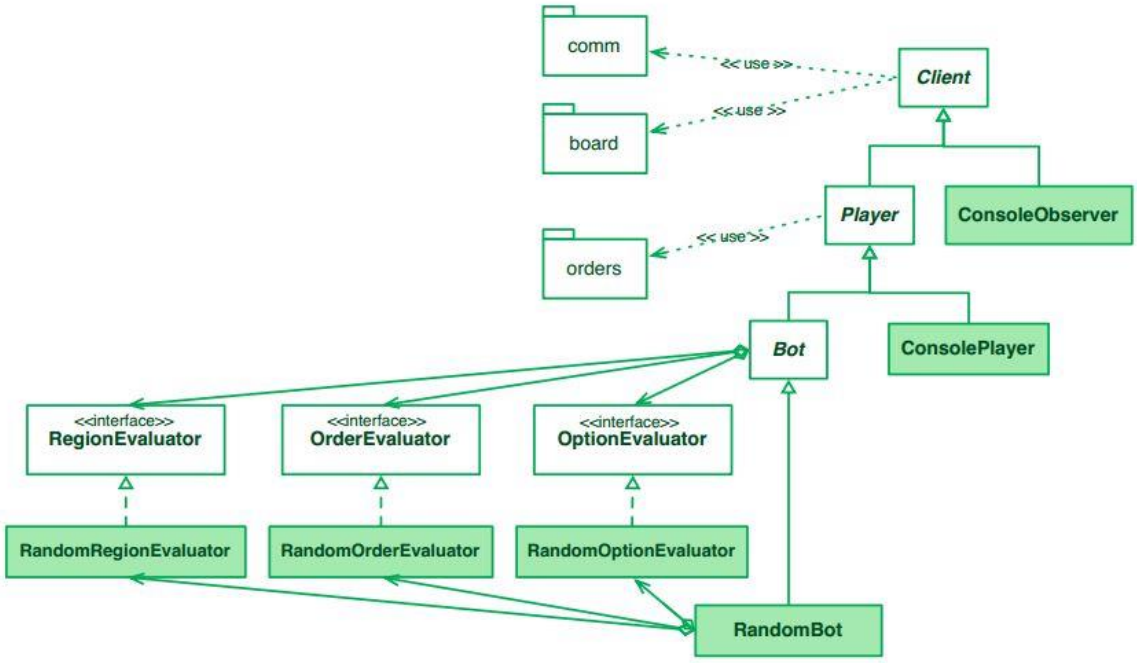


Figura 13. Diseño UML 2.0 de la arquitectura de un agente en DipGame [1].

DipGame ofrece tres evaluadores sobre diferentes aspectos del juego como una región, orden u opción. Cada uno de ellos supone una función de evaluación que el agente usará para sus tomas de decisiones.

En el caso de que el agente quiera negociar con otros, deberá implementar una clase que implemente la interfaz "Negotiator" y una clase responsable de las negociaciones que implemente la interfaz "DipNegoClientHandler". El negociador se ocupa de los mensajes a enviar, y el responsable de las negociaciones de los mensajes a recibir.

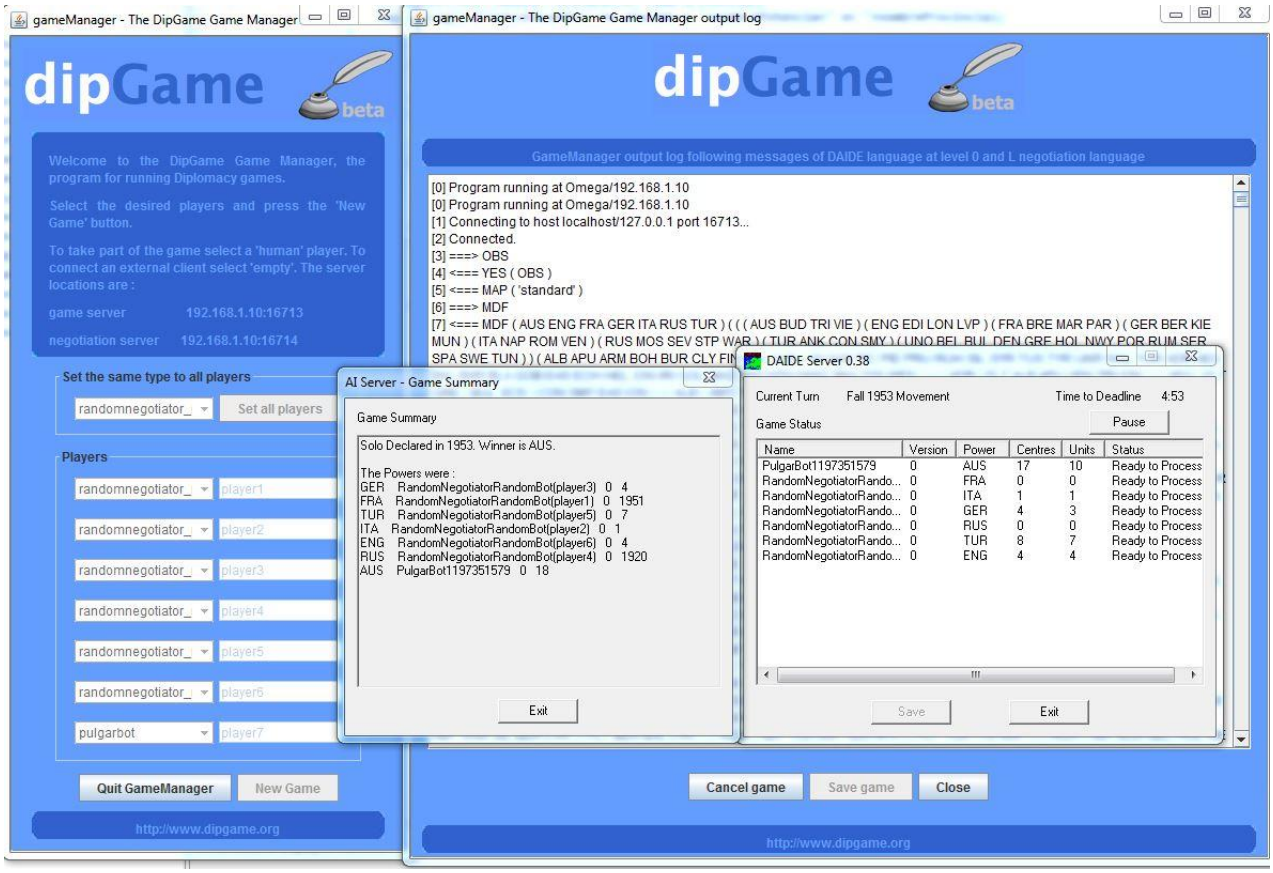


Figura 14. Final de una partida entre agentes con DipGame.

5. PulgarBot

PulgarBot es una implementación de un agente con el framework DipGame. Al usar el framework DipGame, se trata de un agente con una arquitectura BDI (*Belief–desire–intention*), que usa el nivel uno del lenguaje L proporcionado por el framework y negocia con otros agentes peticiones de alianza, tratados de paz y ordenes de apoyo.

Los autores de DipGame simplifican el desarrollo de un agente ya que el framework se ocupa de la búsqueda en el espacio de soluciones, siendo el trabajo del desarrollador la implementación de funciones de evaluación sobre esas soluciones encontradas [19].

El agente escribe en el fichero de registro de salida que maneja el framework. Éste se encuentra en la carpeta “logs” del programa DipGame.

5.1. Estrategia

Al elegir la negociación como principal estrategia, ya que es el framework el encargado de generar las órdenes para nuestras unidades, consideramos como el principal método de victoria la alianza con otras potencias para apoyar nuestros movimientos de unidades.

El agente moverá sus unidades a las provincias mejor valoradas (en forma de movimiento o apoyo) y solicitará a otros agentes peticiones de apoyo para sus unidades.

La estrategia tiene en cuenta que el framework genera órdenes de manera óptima para el agente, en otro caso, el agente puede que no tenga un rendimiento óptimo.

5.2.Arquitectura

PulgarBot extiende la clase *Bot* de DipGame, y requiere implementar tres clases que evalúen las provincias, órdenes y opciones de una partida. Además requiere implementar un negociador de mensajes salientes y un responsable de negociaciones entrantes.

Está basado en la plantilla de clases disponible en la página de DipGame [20], usando como base *RandomBot* y *RandomNegotiator*.

La arquitectura BDI del agente se divide en:

- Creencias
  - Las creencias del agente vienen representadas en la clase *ConocimientoBot*.
- Deseos
  - El deseo principal del agente es ganar la partida. Otros deseos pueden ser la alianza con una potencia, realizar un tratado de paz o solicitar apoyo a otras potencias.
- Intenciones
  - Las intenciones del agente se representan en las órdenes. Las órdenes son evaluadas por el agente y realiza un filtrado en función de su evaluación. El agente seleccionará el conjunto de órdenes mejor evaluado.

5.3. Algoritmo general

DipGame usa los evaluadores del agente para evaluar todas las provincias de la partida. Luego genera una serie de opciones (conjunto de órdenes) en base al estado de la partida. El evaluador de órdenes evalúa estás órdenes y el evaluador de opciones evalúa los conjuntos de éstas (opciones).

El agente recibe una lista de opciones del framework y selecciona la opción a ejecutar durante esa fase.

El agente recibe todas las órdenes que se han realizado en la fase (incluidas sus órdenes) y actualiza su conocimiento. Tras actualizar el conocimiento inicia las

negociaciones con otros agentes, decidiendo qué hacer con los mensajes recibidos y qué mensajes enviar a otros agentes. Tras esto se inicia otra fase del turno.

Cabe destacar que la generación de las órdenes depende totalmente del *framework*, el agente implementado sólo se limita a evaluar el conjunto de órdenes recibido y seleccionar las que considere oportunas.

- Por cada turno de la partida

Algoritmo que realiza el framework:

- Por cada provincia de la partida
  - Evaluar la provincia
- Generar conjuntos de órdenes
- Por cada orden
  - Evaluar la orden individualmente
- Por cada conjunto de ordenes
  - Evaluar el conjunto de ordenes

Algoritmo que realiza el agente:

- Recibir listado de órdenes realizadas por todos los agentes.
  - Actualizar conocimiento
  - Iniciar negociaciones
- Recibir conjunto de órdenes generadas por el framework
  - Seleccionar la mejor opción en base a su valoración

Tabla 5. Algoritmo general del agente.

5.4.Diseño

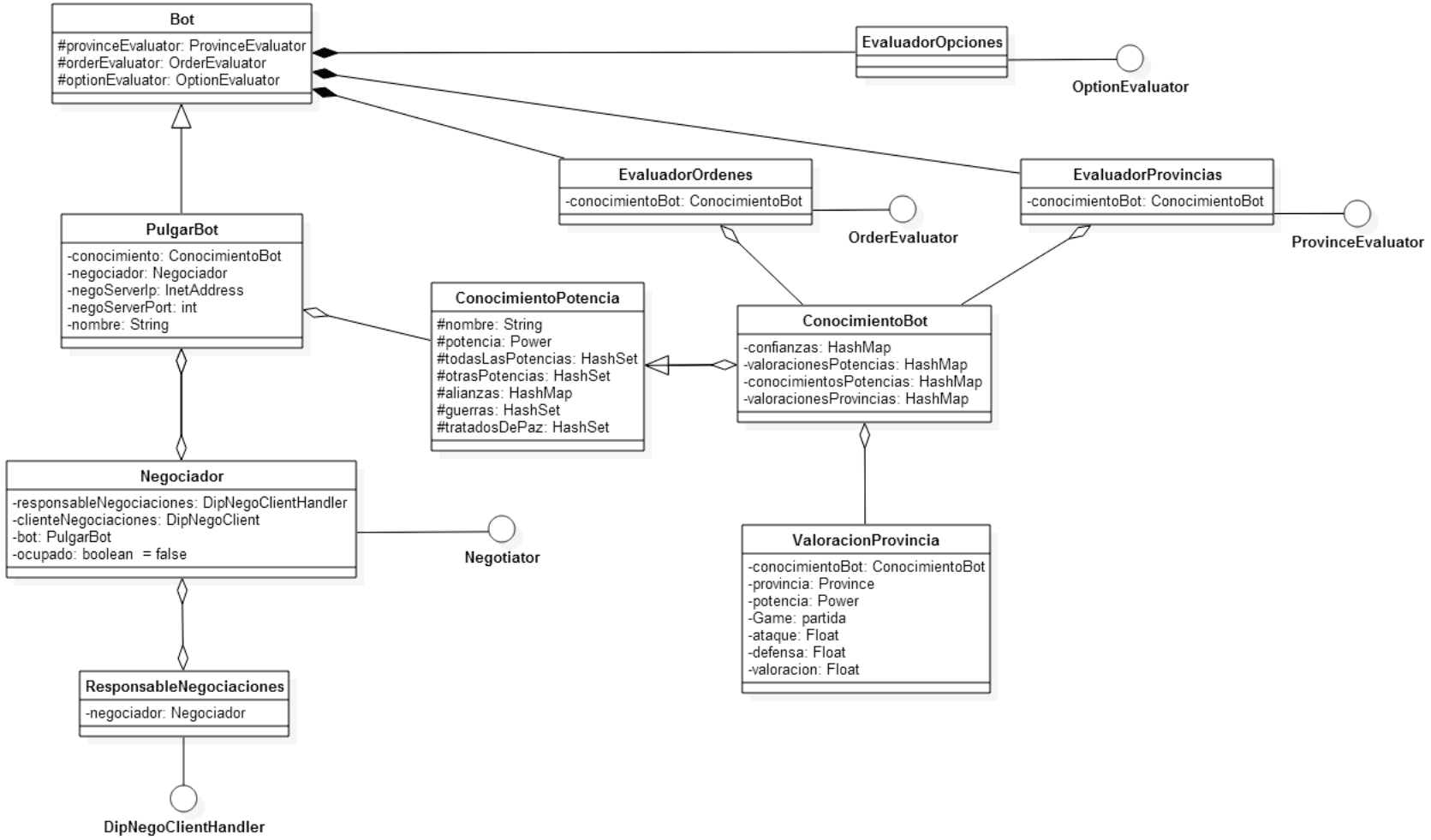


Figura 15. Diseño UML 2.0 de PulgarBot.

5.4.1. PulgarBot

PulgarBot es la clase principal del agente. Contiene el método *main* que inicializa las comunicaciones con el servidor DAIDE.

Se encarga de elegir el conjunto de órdenes por su valoración. Además recibe todas las órdenes efectuadas en cada turno de la partida, pudiendo así actualizar sus creencias sobre el entorno.

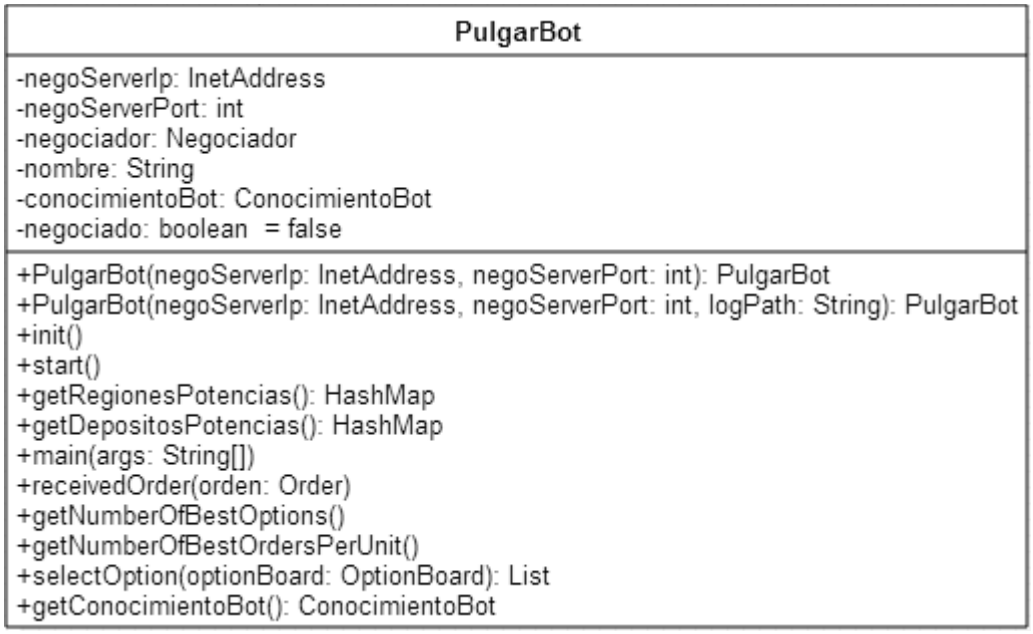


Figura 16. Clase PulgarBot.

5.4.1.1.1. Atributos

Los atributos *negoServerIp* y *negoServerPort* son los atributos del servidor de negociación, la dirección IP y el puerto asociados a él.

El atributo *negociador* es la referencia al componente negociador del agente. Es una instancia de la clase *Negociador* que decide qué peticiones enviar a otros agentes.

El atributo *nombre* indica el nombre del agente, en este caso es “PulgarBot”.

El atributo *conocimientoBot* representa el conocimiento del agente. Es una instancia de la clase *ConocimientoBot* y contiene todas las creencias del agente.

El atributo *negociado* indica si en ese turno el agente ha iniciado las negociaciones con otros agentes.

5.4.1.1.2. Métodos

La clase contiene un constructor para la instanciación del agente, *PulgarBot.* Recibe los datos del servidor de negociación por parámetro. Además puede recibir la ubicación donde se generará el registro de salida de mensajes. También asigna los evaluadores a la superclase.

El método *init i*nicializa el negociador y el conocimiento del agente. Es un método que hay que implementar de la clase *Bot* de DipGame para instanciar el conocimiento y el negociador del agente.

El método *start* se ejecuta al inicio de una partida. Actualiza el conocimiento del agente en ese momento e inicializa el negociador del agente. Además inyecta las dependencias que requieran los evaluadores.

Los métodos *getRegionesPotencias* y *getDepositosPotencias d*evuelven un mapeado de las unidades de todas las potencias y la provincia en la que se encuentran, y un mapeado de los todos depósitos de suministros de la partida y el nombre de la potencia controla dicho depósito, respectivamente.

El método *main* conecta al agente al servidor *DAIDE* que implementa DipGame. A través de él se recibirán y enviarán todas las comunicaciones entre agentes.

El agente en cada fase de turno recibe todas las órdenes realizadas por todas las potencias, incluidas sus propias acciones, con el método *receivedOrder*. Aquí comprueba si ha habido ataques sobre otras potencias y actualiza los conocimientos que tiene sobre dichas potencias. También actualiza las valoraciones que tiene sobre las potencias e inicia las negociaciones si no ha negociado en esa fase. Es un método de la clase *Bot* de DipGame.

Los métodos *getNumberOfBestOptions* y *getNumberOfBestOrdersPerUnit* son métodos que hay que implementar de la clase *Bot* de DipGame. Establecen el número de opciones que generará el framework para el agente y el número de

órdenes que generará el framework para cada unidad. El framework generará 8 opciones para el agente y 4 órdenes para cada unidad.

El método *selectOption* es un método de la clase *Bot* de DipGame que recibe el conjunto de opciones generado por el framework y tras el proceso de evaluación del agente. El agente tiene que devolver la opción elegida a ejecutar, para ello seleccionará la opción que más valoración tenga.

El método *getConocimientoBot* devuelve la instancia del conocimiento del agente.

5.4.2. Conocimiento

El conocimiento del agente es una abstracción de la información que puede observar del entorno de juego. Es la base de creencias del agente según la arquitectura BDI. Se compone de dos tipos de conocimiento: el conocimiento que tiene una potencia de sí misma, y el conocimiento que tiene el agente de sí mismo.

Cada vez que el agente recibe una orden que se ha producido en el juego (sea una orden de él mismo o de otra potencia), el agente actualizará sus creencias.

5.4.2.1. Conocimiento de la potencia

El conocimiento sobre una potencia representa la información relativa a una potencia: su nombre y el objeto de la clase *Power* de DipGame que representa la potencia, además de la lista de alianzas, guerras y tratados de paz asociadas a la misma.

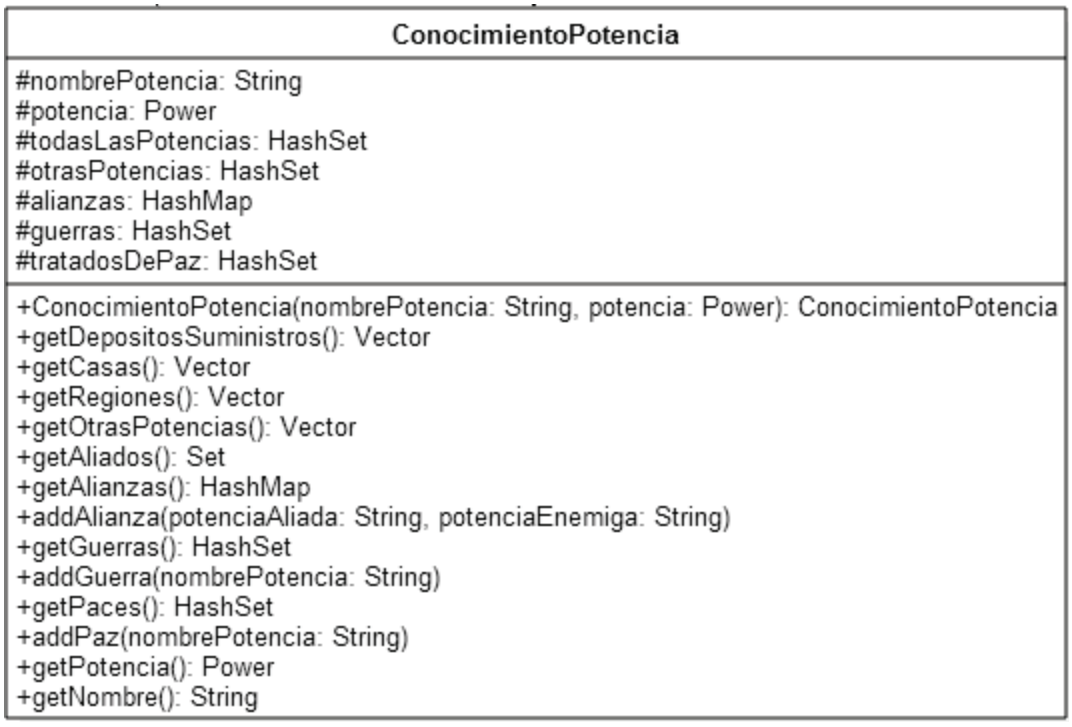


Figura 17. Clase ConocimientoPotencia.

5.4.2.1.1. Atributos

El atributo *nombrePotencia* contiene la abreviación de tres letras del nombre de la potencia.

Potencia	Abreviación
Alemania	GER
Austria	AUS
Francia	FRA
Inglaterra	ENG
Italia	ITA
Turquía	TUR
Rusia	RUS

Tabla 6. Abreviaciones de las potencias en DipGame.

El atributo *potencia* es la instancia de la clase *Power* [21] de DipGame que almacena toda la información relativa a la potencia que representa el agente.

El atributo *todasLasPotencias* representa el conjunto (*HashSet*) de abreviaciones de todas las potencias que participan en una partida, incluida la del propio agente. El atributo *otrasPotencias r*epresenta el conjunto (*HashSet*) de abreviaciones de todas las otras potencias que participan en una partida, excluyendo la potencia que representa el agente (*nombrePotencia*).

El agente representa sus relaciones con otras potencias en los siguientes atributos: alianzas, guerras y tratados de paz.

El atributo *alianzas* representa un mapeado (*HashMap*) entre la abreviación de la potencia aliada y el conjunto (*HashSet*) de enemigos contra los que están aliados.

- Ejemplo:

  Si el conocimiento de la potencia pertenece a Francia (FRA), indica que Francia está aliada con Austria frente a Alemania, Rusia y Turquía.

Clave (Aliado)	Valor (Conjunto de enemigos)
AUS	GER, RUS, TUR

Tabla 7. Ejemplo de mapeado de una potencia contra sus enemigos.

El atributo *guerras r*epresenta el conjunto de abreviaciones de potencias (valor del mapeado) con la que la potencia (clave del mapeado) ha tenido algún conflicto.

- Ejemplo:

  Si Francia atacó a Alemania en alguna provincia, se añadiría Alemania al conjunto de guerras de Francia y Francia al conjunto de guerras de Alemania.

El atributo *tratadosDePaz r*epresenta el conjunto de abreviaciones de potencias con las que se ha aceptado un tratado de paz mediante alguna negociación.

5.4.2.1.2. Métodos

La clase contiene un constructor para la instanciación del conocimiento de la potencia, *ConocimientoPotencia.* Recibe por parámetros la abreviación de la potencia a la que pertenece el conocimiento y el objeto *Power* que representa la potencia. Establece todas las potencias de la partida (atributo *todasLasPotencias*) y las otras potencias excluida la potencia que representa el conocimiento (atributo *otrasPotencias*).

La clase contiene varios métodos para devolver las creencias sobre la potencia:

El método *getDepositosSuministros* devuelve una lista (*Vector*) de abreviaciones de las provincias del mapa que son depósitos de suministros y están controladas por la potencia.

El método *getCasas* devuelve una lista (*Vector*) de abreviaciones de las provincias del mapa que forman parte de las posiciones iniciales de la potencia (Tabla 3).

El método *getRegiones* devuelve una lista (*Vector*) de abreviaciones de las provincias del mapa están controladas por la potencia.

El método *getOtrasPotencias* devuelve una lista (*Vector*) de abreviaciones de las potencias excluyendo a la potencia del conocimiento.

El método *getAliados* devuelve un conjunto (*Set*) de abreviaciones de las potencias que tienen una alianza con la potencia del conocimiento.

El método *getAlianzas* devuelve el mapeado entre la abreviación de la potencia aliada y el conjunto (*HashSet*) de enemigos contra los que están aliados. Devuelve el atributo *alianzas*.

El método *getGuerras* devuelve un conjunto (*Set*) de abreviaciones de las potencias con las que se tiene algún conflicto. Devuelve el atributo *guerras*.

El método *addGuerra a*ñade la abreviación de la potencia recibida por parámetro al conjunto de *guerras*.

El método *getPaces* devuelve un conjunto (*Set*) de abreviaciones de las potencias con las que se tiene un tratado de paz. Devuelve el atributo *tratadosDePaz*.

El método *addPaz* añade la abreviación de la potencia recibida por parámetro al conjunto de *tratadosDePaz*.

El método *getPotencia* devuelve el objeto *Power* (atributo potencia) que representa a la potencia del conocimiento.

El método *getNombre* devuelve la abreviación de la potencia que representa el conocimiento.

5.4.2.2. Conocimiento del agente

El conocimiento del agente expande el conocimiento sobre la potencia, ya que el agente es también una potencia, pero contiene además una lista que indica el nivel de confianza que tiene el agente de las demás potencias, una lista de valoraciones que representa la valoración que el agente atribuye a las demás potencias, una lista de los conocimientos sobre las otras potencias, una lista de las valoraciones de las provincias del juego, una lista de las órdenes que va a realizar y una referencia al objeto *Game* de DipGame.

Figura 18. Clase ConocimientoBot.

5.4.2.2.1. Atributos

El atributo *confianzas* representa una lista de valores enteros que representan la confianza que tiene el agente con esa potencia.

La confianza será un valor del intervalo [-100,100], siendo 0 el valor mínimo que requiere el agente de una potencia para confiar en ella. Al inicio de una partida todas las potencias tendrán un valor de 0.

En el transcurso de las negociaciones o por órdenes de los otros jugadores, el agente modificará la confianza en base a las siguientes acciones:

Acción de guerra en contra nuestra	-25
Tratado de paz aceptado	10
Tratado de paz rechazado	-10
Alianza aceptada	20
Alianza rechazada	-10

Tabla 8. Modificadores a los valores de confianza.

Dichos valores se almacena en la clase Modificador. Es un enumerado *Java* que permite el acceso a cualquier clase de sus atributos.

El atributo *valoracionesPotencias* representa un mapeado entre el nombre de una potencia y el valor entero de la valoración que tiene el agente de esa potencia.

La valoración de una potencia consiste en el número de depósitos de suministros que tiene bajo su control, además de una bonificación extra de cinco puntos si dicho depósito es uno de sus depósitos iniciales.

El atributo *conocimientosPotencias r*epresenta un mapeado (*HashMap*) de la abreviatura de una potencia con el conocimiento que tiene el agente de ella (es un objeto de la clase *ConocimientoPotencia*).

El atributo *valoracionesProvincias re*presenta un mapeado de la abreviatura de una provincia con la valoración que el evaluador de provincias ha hecho de ella. La valoración de una provincia se detalla en el evaluador de provincias.

El atributo *ordenesPorRegion* representa un mapeado de la abreviatura de una región con la orden que se ha comprometido el agente a realizar por medio de una negociación.

La partida del juego se almacena en el atributo *partida.* Referencia al objeto de la clase *Game* de DipGame que contiene la información de la partida, es básicamente el entorno del agente.

5.4.2.2.2. Métodos

La clase contiene un constructor para la instanciación del conocimiento del agente, *ConocimientoBot.* Recibe por parámetros la abreviación de la potencia que representa el agente y el objeto *Power* que representa la potencia. Establece las confianzas sobre las demás potencias a 0 e inicializa los conocimientos que tiene sobre las demás potencias. Además realiza una valoración inicial sobre las demás potencias.

El método *actualizar* actualiza las creencias del agente. Concretamente, actualiza la valoración de otras potencias en ese momento.

Para modificar la confianza respecto a una potencia, el método *cambiarConfianza* modifica la confianza de una potencia con el valor del parámetro recibido, que puede ser negativo.

El método *confiamosMasEn* devuelve *true* si se confía más en la primera potencia recibida por parámetro que en la segunda potencia recibida por parámetro. En otro caso *false*.

El método *getConfianza* devuelve el valor entero que representa el nivel de confianza que tiene el agente de la potencia recibida por parámetro.

El método *getConfianzaOrdenadas* devuelve la lista de valores de confianza ordenada de mayor confianza a menor confianza.

El método *actualizarValoracionesPotencias* actualiza la lista de valoraciones que tiene el agente sobre las demás potencias en base al estado del juego.

El método *getValoracionPotencia* devuelve la valoración de la potencia recibida por parámetro.

El método *getValoracionesPotenciasOrdenadas* devuelve la lista de valoraciones de otras potencias ordenada de mayor valoración a menor valoración.

El método *addValoracionProvincia* añade la valoración de la provincia recibida por parámetro a la lista de valoraciones de provincias.

El método *getValoracionProvincia* devuelve la valoración de la provincia recibida por parámetro.

El método *getConocimientoPotencia* devuelve el conocimiento de la potencia recibida por parámetro.

Para conocer si una potencia es enemiga del agente, tenemos el método *esEnemigaNuestra* que devuelve true si la potencia recibida por parámetro tiene algún conflicto con nuestra potencia o si estamos aliados con alguna potencia en contra de ella.

Si una potencia ataca a otra unidad, el agente añade la potencia atacante a la lista de guerras de la potencia defensora y viceversa, mediante el método *addGuerra*.

El agente tendrá que decidir en las negociaciones si debería aliarse con otra potencia, el método *nosAliamos* devuelve si la potencia a aliar y la lista de enemigos recibidos por parámetro sería una buena alianza. Si la potencia es de nuestra confianza y su valoración más la nuestra es superior a la suma de valoraciones de los enemigos, devuelve *true*. En otro caso devuelve *false*.

Si el agente se alía con una potencia, con el método *addAlianza* añadirá la alianza con la potencia recibida por parámetro y la lista de enemigos recibidos por parámetro a la lista de alianzas.

El agente puede preguntarse si una potencia es aliada nuestra con el método *esAliadaNuestra*. Devuelve *true* si tenemos alguna alianza con la potencia recibida por parámetro. En otro caso, devuelve *false*.

En la fase de negociaciones, el agente puede enviar peticiones de alianza a otras potencias. El método *potenciaParaAlianza* le indica la potencia más favorable para aliarse. Devuelve un mapeado de la potencia elegida para aliarse y del conjunto de potencias de enemigos. La potencia aliada deberá tener igual o más valoración que nuestra potencia y debemos tener algún enemigo en común.

El agente puede preguntarse si una potencia es enemiga de algún aliado nuestro. El método *esEnemigaDeAliado* devuelve *true* si la potencia recibida por parámetro tiene algún conflicto con un aliado. En otro caso, devuelve *false*.

En la fase de negociaciones, el agente puede enviar peticiones de apoyo a otras potencias. Con el método *posiblesApoyos* el agente devuelve un mapeado de las potencias y las unidades que pueden apoyar nuestros movimientos. El agente comprueba las órdenes que tiene en su conocimiento para realizar, y considera las unidades adyacentes al destino de la orden de movimiento como posibles apoyos.

Si el agente firma un tratado de paz con otra potencia, el método *addPaz* añade la paz con la potencia recibida por parámetro a la lista de tratados de paz.

El agente recibirá peticiones de paz de otros agentes y con el método *hacemosPaz* devuelve *true* si el agente considera hacer la paz con la potencia recibida por parámetro o *false* en otro caso. El agente hará la paz con una potencia cuya valoración sea superior a la suma de las valoraciones de las potencias aliadas (incluida su propia valoración). No tiene sentido aceptar la paz de una potencia cuya valoración sea inferior.

En la fase de negociaciones, el agente puede decidir enviar una petición de paz a alguna potencia con la que esté en guerra. El método *potenciasParaPaz* devuelve una lista de potencias que el agente considera aptas para enviar un tratado de paz.

El agente tiene un conocimiento sobre las órdenes que va a realizar. Los métodos *getOrdenRegion*, *setOrdenRegion*, *quitarOrdenRegion* y *eliminarOrdenesRegion* modifican dicho conocimiento.

El método *getOrdenRegion* obtiene la orden que el agente realizará en la región indicara por parámetro. El método *setOrdenRegion* establece una orden para una región. El método *quitarOrdenRegion* quita la orden de una región. El método *eliminarOrdenesRegion* vacía la lista de órdenes actuales.

El método *ordenar* permite ordenar de mayor a menor un mapeado recibido por parámetro por su clave de tipo entero.

El método *getPartida* devuelve la referencia de la clase *Game* de DipGame que representa la partida en la que participa el agente.

5.4.3. Evaluador de provincias

El evaluador de provincias asigna una valoración a cada provincia. Es el principal evaluador del agente, pues de él depende la valoración que reciba una orden.

Figura 19. Clase EvaluadorProvincias.

El evaluador de provincias se basa en la clase *ValoracionProvincia* para realizar las valoraciones. Cada provincia evaluada en el método *evaluate* será añadida al conocimiento del agente.

5.4.3.1.1. Atributos

El evaluador necesita acceder al conocimiento del agente para obtener las valoraciones de las provincias. El atributo *conocimientoBot* es una referencia a dicho conocimiento.

5.4.3.1.2. Métodos

La clase contiene un constructor para la instanciación del evaluador de provincias: *EvaluadorProvincias.*

Para inyectar la dependencia del conocimiento del agente en el evaluador se usa el método *setConocimientoBot*.

El método principal de la clase es *evaluate*. Es el método que obliga a implementar la interfaz *ProvinceEvaluator*. Recibe como parámetros la provincia a evaluar, además de la partida del juego y la potencia que controla la provincia. Asigna la suma del ataque y defensa de la provincia con el método *setValue*.

5.4.4. Valoración de provincias

El agente asigna una valoración a la provincia para evaluar posteriormente sus acciones basándose en ellas.

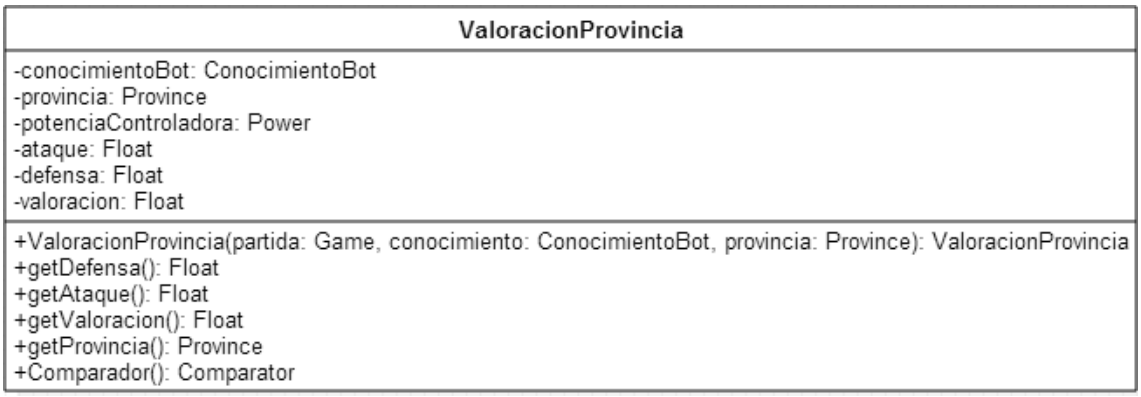


Figura 20. Clase ValoracionProvincia.

5.4.4.1.1. Atributos

Necesita acceder al conocimiento del agente para almacenar las valoraciones de las provincias. El atributo *conocimientoBot* es una referencia a dicho conocimiento.

La valoración pertenece a una provincia, la cuál será referenciada mediante el atributo *provincia*. Es una instancia de la clase *Province* de DipGame.

La provincia puede tener una unidad que esté en ella. En caso de tener una potencia que controle dicha provincia será referenciada en el atributo *potenciaControladora*. Es una instancia de la clase *Power* de DipGame.

La valoración de una provincia se compone de los atributos ataque, defensa y valoración.

El ataque de una provincia indica el valor ofensivo que tiene la potencia que controle el depósito.

Si la provincia no es un depósito de suministro, su ataque será 0.

Si el agente controla dicha provincia, no se considera el ataque de la misma, pues ya la controlamos. Su ataque será 0.

En cambio, si otra potencia controla la provincia, el ataque será el resultado de la siguiente función:

$$ataque = numeroDepositos * numeroDepositos + 4 * numeroDepositos + 16$$

Tabla 9. Valoración del ataque de una provincia.

Siendo *numeroDepositos* el número de depósitos de suministro que controla la potencia.

La defensa de una provincia indica su necesidad de defenderse de enemigos potenciales.

Si la provincia no es un depósito de suministro, su defensa será 0.

Si el agente no controla dicha provincia, no se considera la defensa de la misma, pues no nos interesa defenderla.

En cambio, si la potencia controla la provincia, la defensa de la provincia se considera como el valor de ataque máximo de la potencia más cercana, siendo distinta del agente.

.La valoración de una potencia depende de varios factores:

- Proximidad de otras provincias
- Competición de otras potencias
- Influencia de nuestra potencia alrededor
- Relación con la potencia controladora de la provincia.

La proximidad de otras provincias indica el ataque y defensa de las provincias alrededor de la provincia a evaluar.

Se calcula con una profundidad máxima de 10, es decir sólo tendremos en cuenta la valoración de 10 provincias adyacentes.

Cuanta más próxima sea la provincia, más pesará su valoración pues ésta se reduce cada nivel de profundidad.

$$proximidadActual = ataque + defensa$$

$$proximidad(profundidad) = \frac{(proximidad(profundidad - 1) + proximidadActual)}{5}$$

Tabla 10. Valoración de la proximidad de una provincia.

El factor 5 es la media de provincias vecinas que tiene una provincia.

La competición indica la afluencia de unidades de una potencia alrededor de la provincia. Si una potencia tiene más unidades que otra alrededor de la provincia, se considerará el número de unidades que tenga como competición, restándose a la valoración de la provincia.

La influencia es el valor inverso de la competición. Indica la influencia que tiene nuestra potencia alrededor del a provincia. Por lo tanto, será el número de unidades que tengamos en provincias adyacentes, sumando su valor a la valoración de la provincia.

La valoración de la provincia también depende de la relación que tenga nuestro agente con la potencia que controle la provincia. La valoración se verá modificada según los siguientes valores:

Provincia de aliado	-25
Provincia de enemigo	75
Provincia de un enemigo de un aliado nuestro	50

Tabla 11. Modificadores a la valoración de una provincia.

5.4.4.1.2. Métodos

La clase contiene un constructor para la instanciación de la valoración de la provincia, *ValoracionProvincia.* Recibe por parámetros la abreviación de la potencia que representa el agente y el objeto *Power* que representa la potencia. Establece las confianzas sobre las demás potencias a 0 e inicializa los conocimientos que tiene sobre las demás potencias. Además realiza una valoración inicial sobre las demás potencias.

Incluye varios métodos para devolver los atributos de una provincia. Los métodos *getDefensa*, *getAtaque*, *getValoracion* y *getProvincia* se encargan de ello.

La clase incluye un comparador, un objeto de la clase *Comparator* de Java, que permite comparar dos valoraciones de provincia según su ataque y defensa sumadas. Nos permite ordenar una lista de valoraciones de provincias.

5.4.5. Evaluador de órdenes

El evaluador de órdenes se encarga de asignar una valoración a cada orden recibida. La mayoría de órdenes se realizar en una provincia o hacía una provincia, por lo que su valoración se basará en la valoración de la provincia. Implementa la interfaz *OrderEvaluator* de DipGame.

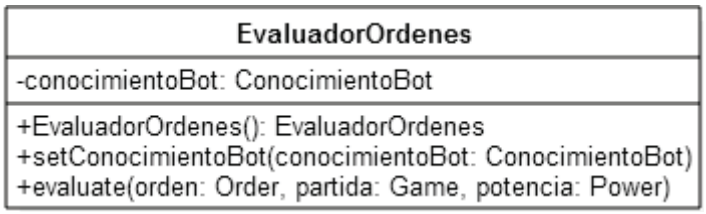


Figura 21. Clase EvaluadorOrdenes.

5.4.5.1.1. Atributos

La clase contiene un constructor para la instanciación del evaluador de órdenes: *EvaluadorOrdenes.*

5.4.5.1.2. Métodos

La clase contiene un constructor para la instanciación del evaluador de órdenes: *EvaluadorOrdenes.*

Para inyectar la dependencia del conocimiento del agente en el evaluador se usa el método *setConocimientoBot*.

El método principal de la clase es *evaluate*. Es el método que obliga a implementar la interfaz *OrderEvaluator*. Recibe como parámetros la orden a evaluar, además de la partida del juego y la potencia que envía la orden. Asigna la valoración a la orden con el método *setValue* según el tipo de orden:

Orden	Evaluación
Construir	Si tenemos más depósitos que unidades: - 50 En otro caso: - -50
Mantener	Defensa de la provincia.
Mover	Valoración de la provincia destino * 1.1
Desbandar	Si tenemos menos depósitos que unidades: - 50 En otro caso: - -50
Retirada	Defensa de la provincia.
Apoyo	Valoración de la provincia destino * 1.2
No Construir (*Waive*)	Si tenemos el mismo número de depósitos que unidades: - 50 En otro caso: - -50

Tabla 12. Evaluación de órdenes.

Las órdenes son generadas por el framework, por lo tanto sólo necesitamos evaluarlas.

5.4.6. Evaluador de opciones

Una opción es un conjunto de órdenes. El evaluador de opciones asigna un valor a la opción recibida. Dado que este evaluador se ejecuta después de evaluar las órdenes individualmente, la valoración de la opción será la suma de las valoraciones de las órdenes que la compongan.

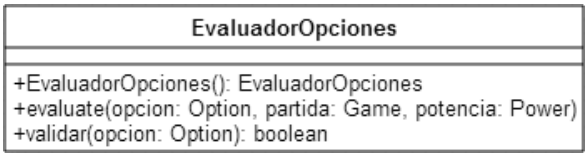


Figura 22. Clase EvaluadorOpciones.

5.4.6.1.1. Métodos

La clase contiene un constructor para la instanciación del evaluador de opciones: *EvaluadorOpciones.*

El método *evaluate* es el método que obliga a implementar la interfaz *OptionEvaluator*. Recibe como parámetros la opción a evaluar, además de la partida del juego y la potencia que asigna la orden.

Al ser las opciones un conjunto de órdenes, previamente generadas por el framework y evaluadas individualmente, la valoración de la opción será la suma de las valoraciones de cada orden que compone la opción.

El método validar filtra las posibles opciones que supongan un conflicto de resolución. Por ejemplo si dos órdenes de dos unidades diferentes tienen una orden de movimiento a la misma provincia, las unidades no se moverían pues sólo una unidad puede ocupar una provincia. Otro ejemplo es si el framework ha generado una orden de apoyo, sin embargo no ha generado la orden de movimiento que apoya la orden, en esto caso sería invalidada.

5.4.7. Negociador

El negociador es la parte del agente que se encarga de negociar y enviar mensajes a otros agentes. Es la clase "Negociador.java" e implementa la clase *Negotiator* de DipGame. El negociador usa el nivel uno del lenguaje L proporcionado por DipGame (Tabla 4).

A la hora de negociar, el negociador evalúa las siguientes posibilidades:

- Hacer la paz con una potencia
- Formar una alianza con una potencia contra otras potencias.
- Pedir una orden de apoyo a otra potencia.

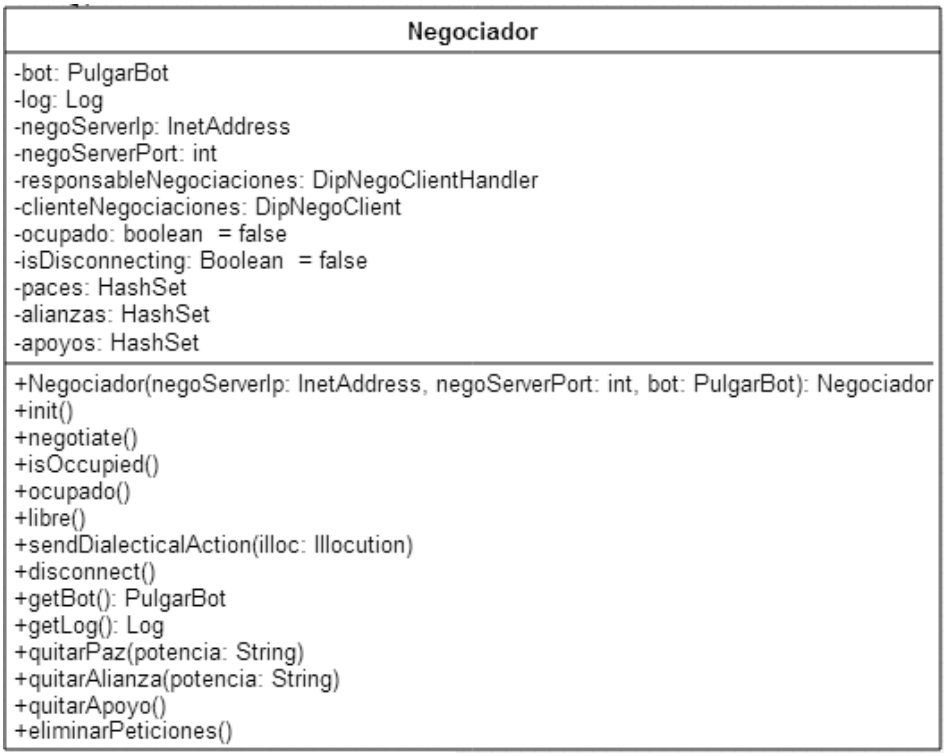


Figura 23. Clase Negociador.

5.4.7.1.1. Atributos

El atributo bot representa el agente al que está asociado este negociador. En este caso sería una instancia de PulgarBot.

El negociador puede imprimir mensajes en el archivo de registro mediante el atributo *log*. Representa la interfaz que tiene DipGame para escribir en el fichero de registro de salida que se genera en la carpeta "logs".

Los atributos *negoServerIp* y *negoServerPort* son los atributos del servidor de negociación, la dirección IP y el puerto asociados a él.

El negociador se encarga de enviar peticiones, pero el encargado de procesar las peticiones entrantes es el atributo *responsableNegociaciones*. Es una instancia de *ResponsableNegociaciones*.

El atributo *clienteNegociaciones* es el cliente de negociaciones de DipGame, con él se envían los mensajes al servidor de negociaciones.

El negociador puede estar ocupado si se encuentra en fase de negociaciones. El atributo *ocupado* representa si el negociador está ocupado (*true*) o libre (*false*).

Cuando el negociador está desconectándose del servidor, el atributo *isDisconnecting* indica si el negociador está desconectándose del servidor (*true*) o no (*false*). Sirve para evitar posibles conflictos en la desconexión que bloquean el programa.

El negociador necesita conocer las peticiones que ha procesado para no enviar la misma petición varias veces en una fase de negociación. Para ello tenemos los atributos *paces*, *alianzas* y *apoyos* que contienen las abreviaciones de potencias a las que se le haya enviado una petición. Las potencias no se quitarán hasta recibir una contestación de ellas o pasar el turno del juego.

5.4.7.1.2. Métodos

La clase contiene un constructor para la instanciación del negociador: *Negociador*. Crea una instancia del negociador y recibe la dirección IP del servidor de negociaciones, y el puerto para establecer la conexión, además del agente del que depende el negociador.

El negociador se inicializa con el método *init*. Instancia el cliente de las negociaciones y conecta al servidor de negociaciones. Es un método de la interfaz *Negotiator*.

El método *negotiate* es el método más importante del negociador. Le permite iniciar el proceso de negociación del negociador. Es un método de la interfaz *Negotiator* de DipGame. El algoritmo de negociación se puede observar en la Tabla 13.

El método *isOccupied* es un método de la interfaz *Negotiator* de DipGame. Devuelve *true* si el negociador está ocupado, en otro caso *false*.

Para establecer el estado del negociador se utilizan los métodos *ocupado* y *libre*. El método ocupado establece el atributo *ocupado* a *true* y el método libre establece el atributo a *false*.

El método encargado de enviar los mensajes al cliente de negociaciones es *sendDialecticalAction*. Envía el mensaje recibido al cliente para su posterior envío al servidor de negociaciones.

El método *disconnect* desconecta al negociador del servidor de negociaciones.

Los métodos *getBot* y *getLog* devuelven las instancias del agente y la interfaz del registro de salida, respectivamente.

Si se ha recibido alguna contestación a las peticiones enviadas, el negociador eliminará a la potencia de la lista de *paces*, *alianzas* o *apoyos* según sea el tipo. Para ello tiene los métodos *quitarPaz*, *quitarAlianza* y *quitarApoyo*.

- **¿Hacemos la paz con alguna potencia?**
  - Solicitar al agente una lista de potencias para realizar la petición de paz.
  - Si no está en la lista de potencias a las que se le ha enviado una petición de paz:
    - Solicitar la paz a dicha potencia.
- **¿Nos aliamos con alguna potencia?**
  - Solicitar al agente una lista de potencias para realizar la petición de alianza.
  - Si no está en la lista de potencias a las que se le ha enviado una petición de alianza:
    - Solicitar la alianza a dicha potencia.
- **¿Pedimos alguna orden de apoyo?**
  - Solicitamos al agente una lista de posibles apoyos.
  - Solicitamos al agente las órdenes actuales.
  - Por cada potencia que nos pueda apoyar:
    - Si no está en la lista de potencias a las que se le ha enviado una petición de apoyo:
      - Comprobar si una orden actual contiene alguna región que pueda tener apoyo.
        - Enviar solicitud de apoyo.

Tabla 13. Algoritmo de negociación.

5.4.8. Responsable de las negociaciones

El responsable de las negociaciones se encarga de dar respuesta a los mensajes recibidos por otras potencias. Es la clase "ResponsableNegociaciones.java" e implementa la clase *DipNegoClientHandler* de DipGame. Al igual que el negociador usa el nivel uno del lenguaje L.

Figura 24. Clase ResponsableNegociaciones.

5.4.8.1.1. Atributos

El atributo *negociador* representa el negociador del agente, ya que el responsable de las negociaciones también negocia los mensajes recibidos, y por lo tanto requiere poder enviar contestaciones a otros agentes.

5.4.8.1.2. Métodos

La clase contiene un constructor para la instanciación del responsable de las negociaciones: *ResponsableNegociaciones*. Crea una instancia del responsable y recibe la referencia al negociador del que depende.

La interfaz *DipNegoClientHandler* requiere implementar varios métodos: handleClientAccepted, handleErrorMessage, handleFirstGamePhase, handleNewGamePhase, handleNegotiationMessage y handleServerOff. El método más relevante es *handleNegotiationMessage* pues es el encargado de tratar el mensaje recibido por parámetro. Según el tipo de mensaje, que puede ser una propuesta, admisión o rechazo, se delega en otro método para tomar una decisión.

Si el mensaje recibido es una propuesta, el método *comprobarPropuesta* se encarga de comprobarla. En el nivel uno de la Figura 12 se muestra que una propuesta puede ser de tipo *AGREE* y *COMMIT*.

En caso de ser un acuerdo (*agree*), se comprueba el tipo de acuerdo, el cuál puede ser: alianza, tratado de paz o realizar una orden. Por algún error no se pueden tratar las propuestas de compromiso (*commit*) – Ver Conclusiones -.

Si la propuesta es una alianza, delegamos en el conocimiento del agente la aceptación o rechazo de la alianza (método *nosAliamos* de *ConocimientoBot*).

Si la propuesta es un tratado de paz, delegamos en el conocimiento del agente la aceptación o rechazo del tratado (método *hacemosPaz* de *ConocimientoBot*).

Si la propuesta es hacer una orden, comprobamos si la valoración de la orden que tiene el agente para esa región es inferior a la orden de la propuesta recibida, en tal caso se actualiza la orden para dicha región en el conocimiento del agente.

En caso de que el mensaje sea una admisión, es decir, otro agente ha aceptado una propuesta que hayamos realizado previamente, el método *comprobarAdmision* será el encargado de procesarla.

En el caso de ser una alianza, se añadirá la alianza al conocimiento del agente y se modificará la confianza positivamente.

En el caso de ser un tratado de paz, se añadirá el tratado de paz al conocimiento del agente y se modificará la confianza positivamente.

En el caso de ser la realización de una orden, se añadirá la orden al conocimiento del agente según la región en la que se desarrolle.

En caso de que el mensaje sea un rechazo, es decir, otro agente ha rechazado una propuesta que hayamos realizado previamente, el método *comprobarRechazo* será el encargado de procesarlo.

En el caso de ser una alianza, se modificará la confianza negativamente.

En el caso de ser un tratado de paz, se modificará la confianza negativamente.

En el caso de ser la realización de una orden, no se realizará ninguna modificación.

Al recibir una propuesta, el agente puede decidir aceptarla o rechazarla. Los métodos *rechazarTrato* y *aceptarTrato* son los encargados de generar mensajes de admisión o rechazo. En caso de querer rechazar un trato, se le indica al negociador que se quiere enviar un mensaje de rechazo (clase *Reject* de DipGame). En caso de querer aceptar un trato, se le indica al negociador que se quiere enviar un mensaje de admisión (clase *Accept* de DipGame).

6. Pruebas y resultados

Probar el rendimiento de un agente en Diplomacy es difícil porque no tenemos una función de evaluación bien definida. Al requerir las partidas siete jugadores, se puede considerar que un agente es mejor que otro si ha acabado en mejor posición (ganando la partida o por el número de depósitos de suministros). Medir el rendimiento sólo por la victoria de una partida es injusto porque no valora la supervivencia del agente en el entorno, un agente que sobreviva todas las partidas (acabe la partida con al menos una unidad) será mejor que un agente que quede eliminado en todas las partidas. Por ello, y como en diferentes deportes como el fútbol, en el cuál un empate otorga un punto frente a los tres de una victoria, se ha diseñado un sistema de clasificación que evalúe de forma relativa el rendimiento del agente:

Resultado	**Puntos**
Ganador de la partida	10
Más de 10 depósitos controlados	7
Más del número de depósitos inicial	3
Menos o igual que el número de depósitos inicial	1
Eliminado	0

Tabla 14. Sistema de clasificación para evaluar un agente.

Discernir entre el número de depósitos de suministros con los que finaliza una partida es importante porque algunos agentes pueden tener la suerte de sobrevivir gracias a su posición inicial. Además la distancia entre la potencia ganadora y ésta aumenta sin duda la probabilidad de sobrevivir de una potencia.

Para los experimentos del agente, se ha usado uno de los agentes incluidos en DipGame que permite negociación: R*andomNegotiatorRandomBot* en su versión 1.3. Se ha usado este agente porque los demás no soportan negociación que a priori

es el punto fuerte de este agente, además de la imposibilidad de encontrar otros agentes de terceras partes y creados a partir de DipGame que soporten negociación. No se ha optado por evaluar al agente contra sí mismo pues el objetivo es comprobar su rendimiento frente a otro agente base. PulgarBot está basado en el esqueleto de clases de R*andomNegotiatorRandomBot*, pero difiere en la manera de evaluar el entorno de la partida. Las órdenes son generadas en ambos casos por el framework.

El bot R*andomNegotiatorRandomBot* responde y realiza negociaciones de manera aleatoria, además de evaluar las provincias, órdenes y opciones de manera aleatoria. Esto no quiere decir que sea totalmente aleatorio, pues al ser las órdenes generadas por el framework, es posible que el agente seleccione la orden más conveniente si el framework es capaz de generarlas óptimamente. Destacar que ambos agentes sólo evalúan las órdenes, pues la diferencia entre ellos será el método de evaluación.

Debido a la naturaleza del juego, para evaluar enfrentamientos entre 2 agentes necesitamos distribuir de manera equitativa los agentes entre los siete jugadores de una partida. Por ello, se diseñan dos escenarios de partidas:

- Un escenario tendrá 20 partidas y enfrentará a 4 instancias de PulgarBot contra 3 instancias de R*andomNegotiatorRandomBot*.
- Otro escenario tendrá 20 partidas y enfrentará a 3 instancias de PulgarBot contra 4 instancias de R*andomNegotiatorRandomBot*.

Las partidas han sido configuradas con los parámetros por defecto. Debido a que el servidor asigna las potencias aleatoriamente, y para mantener una diversidad de potencias en los resultados, ha sido necesario suspender manualmente algunas partidas en el inicio.

Partidas PulgarBot (4) contra RandomNegotiatorRandomBot(3)							
Potencia	**Ganador**	**+10**	**+Inicio**	**-=Inicio**	**Eliminado**	**Partidas**	**Puntos**
Alemania	5	2	1	1	1	10	68
Austria	6	1	2	0	2	11	73
Francia	6	0	3	1	1	11	70
Inglaterra	3	0	2	7	0	12	43
Italia	5	1	3	2	1	12	68
Rusia	8	1	1	0	0	10	90
Turquía	4	4	2	2	2	14	76
Total	37	9	14	13	7	80	488

Tabla 15. Resultados de PulgarBot contra RandomNegotiatorRandomBot.

Partidas RandomNegotiatorRandomBot(4) contra PulgarBot (3)							
Potencia	**Ganador**	**+10**	**+Inicio**	**-= Inicio**	**Eliminado**	**Partidas**	**Puntos**
Alemania	4	1	2	1	3	11	54
Austria	3	1	1	3	4	12	43
Francia	4	0	3	2	2	11	51
Inglaterra	1	0	2	6	1	10	22
Italia	3	1	1	4	2	11	44
Rusia	5	2	0	3	3	13	67
Turquía	3	1	1	6	1	12	46
Total	23	6	10	25	16	80	327

Tabla 16. Resultados de RandomNegotiatorRandomBot contra PulgarBot.

En los resultados de las veinte partidas contra el agente se observa que algunas potencias logran mejores resultados que otras. En el caso de Inglaterra, es la potencia que más cuesta conseguir la victoria, al ser una potencia marítima que sin el

soporte de la orden de convoy en DipGame le permite sobrevivir sin mucha dificultad, pero no le permite establecer un control de depósitos en el centro de Europa debido a su falta de transporte. La potencia que más fácil tiene conseguir la victoria es Rusia, puede que sea a su control inicial de más depósitos de suministros que otras potencias.

El agente PulgarBot ha conseguido un 60% más de victorias que su oponente. Además, en el cómputo global de puntos tiene un 49% más de puntos, lo que le otorga un mejor rendimiento sobre el agente aleatorio.

Potencia	**PulgarBot**	**RandomNegotiatorRandomBot**
Alemania	6,8	4,9
Austria	6,6	3,6
Francia	6,3	4,6
Inglaterra	3,6	2,2
Italia	5,6	4
Rusia	9	5,2
Turquía	5,4	3,8
Media	6,2	4

Tabla 17. Puntuación media de cada potencia por partida.

En términos generales, el agente PulgarBot consigue de media un 55% más de puntos que el agente *RandomNegotiatorRandomBot*.

7. Conclusiones

Tras la inspección de los archivos de registro generados por el framework, se observa que el agente PulgarBot no puede aprovechar su capacidad de negociación ya que el comportamiento aleatorio de algunos adversarios no le permite realizar unas negociaciones productivas. En cambio, las negociaciones entre el mismo tipo de agentes producen una incapacidad de traicionar a sus aliados, esto supone una pérdida de rendimiento ya que las unidades se quedan bloqueadas sin poder acceder a otras provincias al no querer atacar una provincia aliada. Esto se observa con mayor detalle en Turquía, la segunda peor potencia evaluada tras Inglaterra.

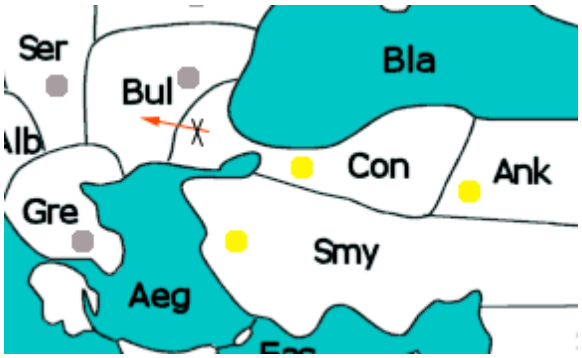


Figura 25. Unidad en Constantinopla bloqueada.

Al no permitir la orden de convoy, una unidad en Constantinopla quedará bloqueada si no decide nunca atacar a una unidad que permanezca en Bulgaria.

En el caso de Inglaterra, sólo se pueden permitir conquistar los depósitos que se encuentren en la costa debido a sus dos flotas iniciales. La unidad terrestre de Liverpool no podrá moverse sin tener un convoy a Europa. Los depósitos más cercanos a Inglaterra tienen una fuerte disputa al encontrarse en el centro de Europa, y estar rodeados por Alemania, Francia y Austria.

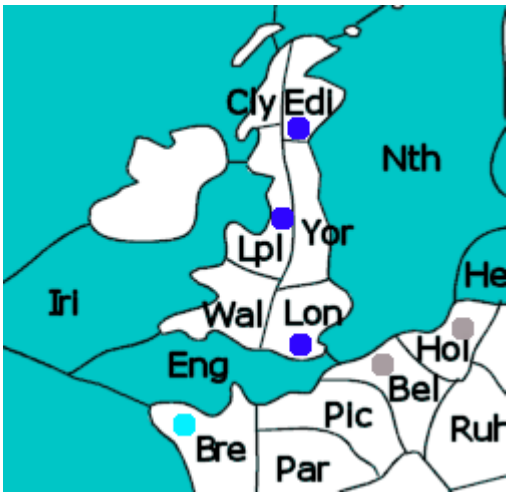


Figura 26. Situación de Inglaterra.

Las peticiones realizadas por el agente que son respondidas con una admisión no son cumplidas en la mayoría de casos, ya que no están obligadas por algún medio a cumplirlas, y por lo tanto tienen un valor simbólico. Al no permitirse empates, la victoria solitaria es muy difícil de conseguir si la estrategia principal es la negociación. Por este motivo, y basándose en la generación de órdenes por parte del framework, es necesaria una revisión de la estrategia que sitúe la evaluación del entorno como principal fuerte del agente.

El objetivo del trabajo era implementar un agente en el framework DipGame y que sea capaz de negociar con otros agentes en el nivel uno del lenguaje L. Dadas las pruebas y resultados, se puede concluir que el agente supone una mejora de alrededor del 50% sobre el agente aleatorio.

7.1. Mejoras y trabajos futuros

Un agente software ofrece mucha flexibilidad en su implementación, de esta manera el agente puede aprovechar las múltiples técnicas de inteligencia artificial disponibles para mejorar su rendimiento. A pesar de que a priori no se podría estimar el rendimiento de la técnica, las siguientes técnicas suponen mejoras prometedoras:

- Añadir nuevos niveles de comunicación proporcionados por el framework (Figura 11). De esta manera se mejorarían las capacidades de negociación, algo vital en un agente de DipGame ya que la búsqueda de soluciones en el espacio de órdenes posibles por turno depende totalmente del framework, delegando en el agente la capacidad de negociación como punto fuerte. Los niveles 2 y 3 del lenguaje L de DipGame supondrían una gran mejora en el conocimiento del agente sobre otras potencias, mejorando su capacidad de tomar decisiones.
- Aplicar un algoritmo de aprendizaje que permita a los evaluadores del agente basarse en entrenamientos previos a partidas. Además de otras técnicas de inteligencia artificial que permitan predecir los futuros movimientos de otros agentes.
- Aplicar un entrenamiento mediante *Q-Learning* para conseguir una política de acciones por cada potencia, de esta manera el agente podrá consultar la política para evaluar las acciones recibidas por el framework.
- Optimizar los parámetros que modifican los valores de confianza y otras creencias del agente. Se podría usar un algoritmo genético para determinar los parámetros óptimos de las funciones de evaluación usadas por el agente.
- Usar un modelo de reputación. La reputación supone una percepción de la confianza que tienen los otros agentes sobre uno. En el caso de usar niveles de lenguaje superiores, el agente podría considerar si la información facilitada por otro agente es cierta o falsa según su reputación.

7.2. Problemas encontrados

El framework requiere una documentación del código más extensa. Únicamente se tiene acceso a la documentación *Javadoc* del framework sin comentarios [22] y a los trabajos realizados por los autores sobre el framework [1] [19], los cuales profundizan más en la teoría que en el framework en sí.

En el nivel uno del lenguaje L ofrecido por el framework, uno de los mensajes disponibles es comprometerse a realizar una acción (oferta de tipo *Commit*), sin embargo, no se permite promocionar el objeto *Offer* a *Commit*, por lo que no se pueden tratar las propuestas recibidas por otros agentes de este tipo.

La imposibilidad de encontrar un bot inteligente implementado en DipGame para probar el agente del trabajo, por lo que el único bot contra el que se puede probar el agente es R*andomNegotiatorRandomBot*.

La dificultad de realizar experimentos automáticos con el banco de pruebas. Según la página del framework se ofrece la posibilidad de lanzar el banco de pruebas por línea de comandos:

"We make easy to implement an automatic game launcher allowing the execution of gameManager from a terminal passing as parameters the players that should take part in a game." [23]

Sin embargo no se explica el orden de parámetros, y la aplicación no responde a parámetros por línea de comandos.

8. Bibliografía

[1] A. Fabregues y C. Sierra, «DipGame: a challenging negotiation testbed,» *Engineering Applications of Artificial Intelligence,* vol. 24, nº 7, p. 1137–1146, Octubre 2011.

[2] M. Wooldridge y N. Jennings, «Intelligent Agents: Theory and Practice,» *Knowledge Engineering Review,* vol. 10, nº 2, 1995.

[3] S. . J. Russell y P. Norvig, *Artificial Intelligence: A Modern Approach*, 1994.

[4] H. S. Nwana , «Software Agents: An Overview,» *Knowledge Engineering Review,* vol. 11, nº 3, pp. 1-40, 1996.

[5] W. Brenner, R. Zarnekow y H. Wittig, *Intelligent Software Agents: Foundations and Applications*, Springer, 1998.

[6] M. Bratman, *Intention, plans, and practical reason*, Harvard University Press, 1987.

[7] R. S. Anand y M. P. Georgeff , «BDI Agents: From Theory to Practice,» de *Proceedings of the First International Conference on Multiagent Systems*, 1995.

[8] J. Ferber, *Multi-Agent System: An Introduction to Distributed Artificial Intelligence*, Harlow: Addison Wesley Longman, 1999.

[9] N. Jennings, «On Agent-Based Software Engineering,» *Artificial Intelligence,* vol. 117, nº 2, pp. 277-296, 2000.

[10] M. Wooldridge, *An Introduction to MultiAgent Systems*, New York: John Wiley & Sons, Inc., 2001.

[11] S. Appelcline, Designers & Dragons, Mongoose Publishing, 2011.

[12] C. E. Shannon, «XXII . Programming a Computer for Playing Chess,» *Philosophical Magazine, Ser. 7,* vol. 41, nº 314, pp. 256-75, Marzo 1950.

[13] S. J. Johansson, «On using Multi-agent Systems in Playing Board Games,» de *International Conference on Autonomous Agents and Multiagent Systems 2006*, 2006.

[14] A. B. Calhamer, «Diplomacy Rules,» [En línea]. Disponible: http://www.wizards.com/avalonhill/rules/diplomacy.pdf. [Último acceso: 28 5 2014].

[15] Diplomacy AI Development Environment, «Diplomacy AI Development Environment,» 16 5 2014. [En línea]. Disponible: http://www.daide.org.uk/. [Último acceso: 16 5 2014].

[16] Diplomacy AI Development Environment, «Diplomacy AI Development Environment Message Syntax,» 18 5 2014. [En línea]. Disponible: http://www.ellought.demon.co.uk/dipai/daide_syntax.pdf. [Último acceso: 18 5 2014].

[17] Diplomacy AI Development Environment, «News - DAIDE,» [En línea]. Disponible: http://daide.org.uk/index.php?title=News#Level_30_version_of_Albert_released. [Último acceso: 20 5 2014].

[18] J. v. Hal, «Diplomacy AI - Albert,» [En línea]. Disponible: https://sites.google.com/site/diplomacyai/albert. [Último acceso: 21 5 2014].

[19] A. Fabregues y C. Sierra, «A Testbed for Multiagent Systems Technical Report IIIA-TR-2009-09,» IIIA: Institut d'Investigaci en Intelligncia Artificial, Barcelona, 2009.

[20] «dipgame - dipBots page,» [En línea]. Disponible:

http://www.dipgame.org/browse/examples. [Último acceso: 2014 04 20].

[21] «Power,» [En línea]. Disponible: http://www.dipgame.org/downloads_nou/dip-1.6-javadoc/es/csic/iiia/fabregues/dip/board/Power.html.

[22] «DipGame Javadoc,» [En línea]. Disponible: http://www.dipgame.org/downloads_nou/dip-1.6-javadoc/. [Último acceso: 01 06 2014].

[23] «dipgame - GameManager,» [En línea]. Disponible: http://www.dipgame.org/browse/gameManager. [Último acceso: 21 04 2014].

[24] «Java Downloads for All Operating Systems,» [En línea]. Disponible: http://www.java.com/en/download/manual.jsp. [Último acceso: 22 04 2014].

[25] «How do I set or change the PATH system variable?,» [En línea]. Disponible: http://www.java.com/en/download/help/path.xml. [Último acceso: 22 4 2014].

[26] «XVI CONVENIO COLECTIVO ESTATAL DE EMPRESAS DE CONSULTORÍA Y ESTUDIOS DE MERCADOS Y DE LA OPINIÓN PÚBLICA» [En línea]. Disponible: https://www.boe.es/boe/dias/2009/04/04/pdfs/BOE-A-2009-5688.pdf.

Anexo I: Instalación de DipGame

Para ejecutar el banco de pruebas DipGame en *Windows* se necesita [23]:

- Java 6 [24]
- AiServer
  - http://www.ellought.demon.co.uk/dipai/aiserver.msi
- AiMapper
  - http://www.ellought.demon.co.uk/dipai/aimapper.msi
- gameManager
  - http://www.dipgame.org/downloads_nou/gameManager-1.0.zip

Una vez instalado Java y establecida la configuración del *PATH* [25], procedemos a instalar las demás dependencias.

- AiServer

Ejecutamos aiserver.msi:

Figura 27. Instalación del servidor DAIDE.

Hacemos clic en *Next*:

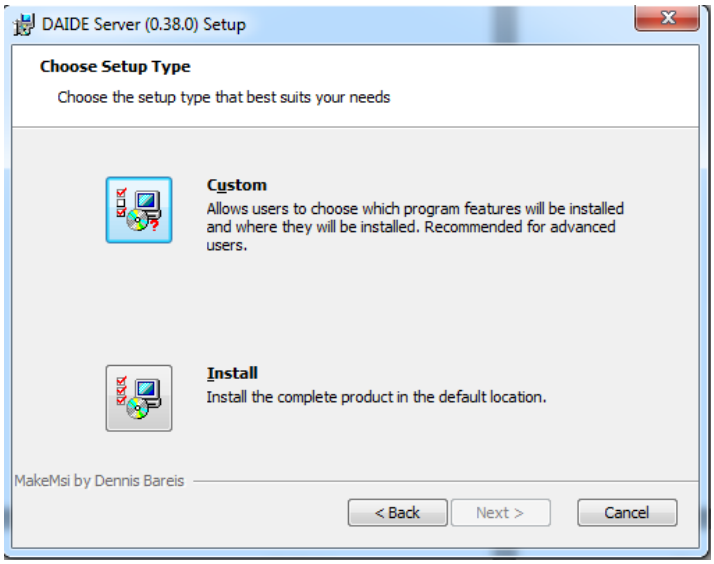


Figura 28. Instalación del servidor DAIDE (2).

Seleccionamos *Install*, y en la siguiente ventana *Install*.

Se iniciará la instalación. Una vez completada podremos finalizarla (*Finish*).

- AiMapper

Ejecutamos aimapper.msi:

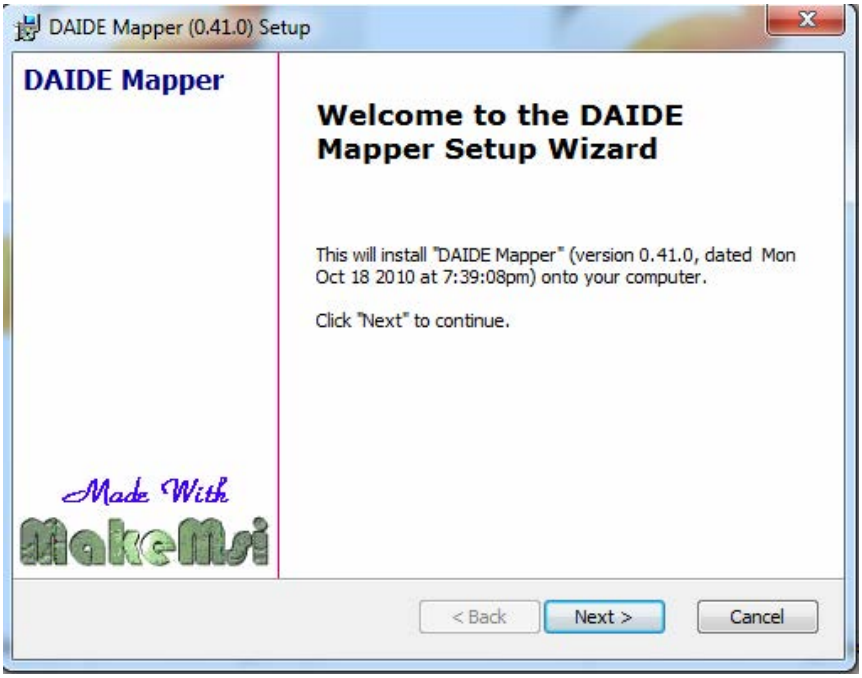
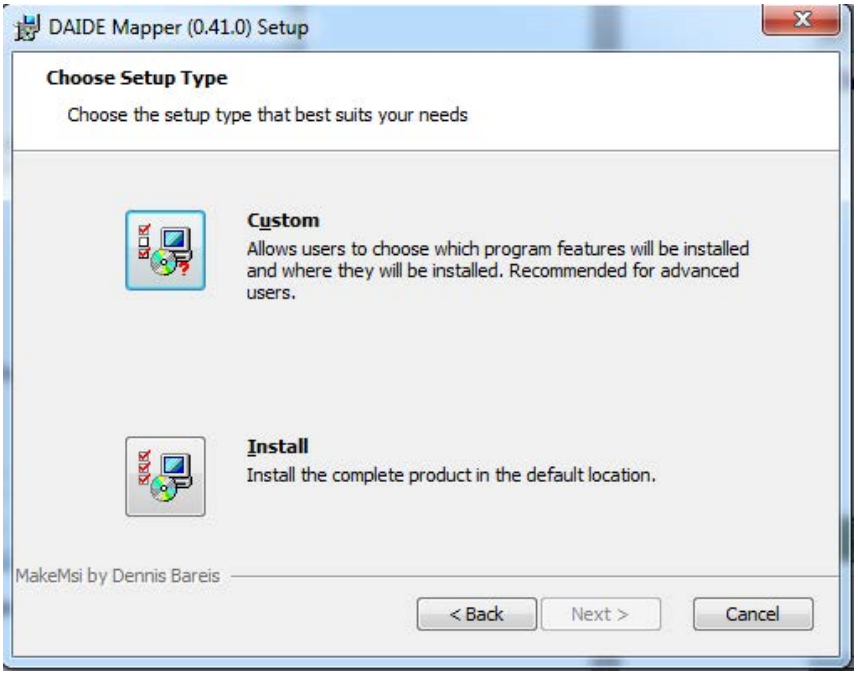


Figura 29. Instalación de DAIDE Mapper.

Hacemos clic en *Next*:



Seleccionamos *Install*, y en la siguiente ventana *Install*.

Se iniciará la instalación. Una vez completada podremos finalizarla (*Finish*).

- gameManager

Descomprimimos el archivo "gameManager-1.0.zip" en una carpeta:

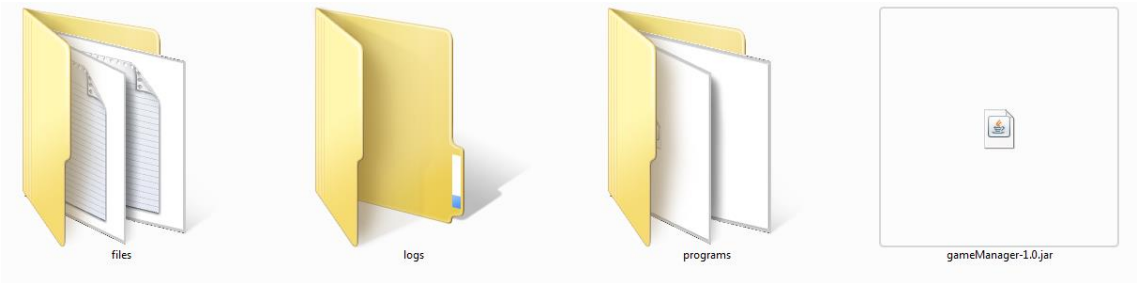


Figura 30. Ubicación del gameManager.

Si tenemos la extensión *jar* asociada a Java podemos ejecutar "gameManager-1.0.jar" que nos recibirá con la ventana del banco de pruebas:

Figura 31. Ventana del Game Manager de DipGame.

Anexo II: Compilación del agente

Para compilar un agente en un archivo *jar* con el objetivo de usarlo en el banco de pruebas de DipGame se necesita:

- Apache Ant
  - https://ant.apache.org/
  - https://code.google.com/p/winant/ (Windows)

DipGame incluye en sus plantillas de bots un fichero build.xml que permite la compresión de un agente de forma sencilla y rápida en un fichero *jar*.

```
<project name="PulgarBot-1.0" default="export" basedir=".">
    <description>
        Archivo de generacion de PulgarBot.
    </description>
  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="bin"/>
  <property name="dist"  location="export"/>
  <property name="lib"  location="lib"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}"/>
    <!-- Create the lib directory -->
    <mkdir dir="${lib}"/>
  </target>

  <target name="compile" depends="init" description="compile the source " >
    <!-- concatenation of lib paths for classpath -->
    <path id="lib.classpath">
      <fileset dir="lib/" includes="**/*.jar" />
    </path>
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}" classpathref="lib.classpath"/>
```

Figura 32. Archivo build.xml para la compresión del agente.

El archivo build.xml se encuentra en la carpeta raíz del código del agente:

Figura 33. Ubicación del archivo build.xml.

Desde la línea de comandos se ejecuta el comando *ant* en la ubicación del archivo para generar el archivo *jar*.

```
D:\Dropbox\Dip\PulgarBot>ant
Buildfile: D:\Dropbox\Dip\PulgarBot\build.xml

init:

compile:
    [javac] D:\Dropbox\Dip\PulgarBot\build.xml:28: warning: 'includeantruntime' was not set, defaulting to build.sysclasspath=last; set to false for repeatable builds

dist:
      [jar] Building jar: D:\Dropbox\Dip\PulgarBot\export\PulgarBot-1.0.jar

export:
      [jar] Building jar: D:\Dropbox\Dip\PulgarBot\export\PulgarBot-1.0.jar

BUILD SUCCESSFUL
Total time: 0 seconds

D:\Dropbox\Dip\PulgarBot>
```

Figura 34. Generación del archivo jar del agente.

Esto genera el archivo jar en la carpeta *export*:

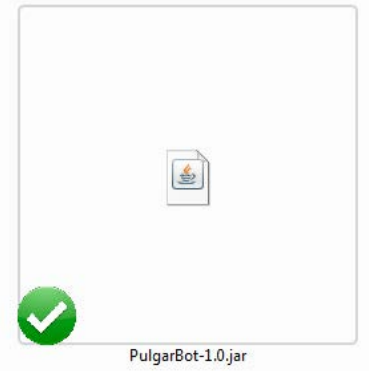


Figura 35. Archivo jar del bot.

Este archivo *jar* será el usado por DipGame para cargar el agente en el programa.

Anexo III: Ejecución de un agente

Nos situamos en la carpeta donde se descomprimió *gameManager-1.0.zip*:

Figura 36. Directorio de gameManager.

En la carpeta "files" necesitamos modificar el archivo "availablePlayers.txt" para añadir el agente a instalar:

```
//PLAYERS AND EXECUTION COMANDS TO RUN THEM.
//The player menus are populated with these players in the given order
dumbbot;<JAVA_ENV> -jar programs/DumbBot-1.3.jar <ip> <port> <name> <log_folder>
randombot;<JAVA_ENV> -jar programs/RandomBot-1.3.jar <ip> <port> <name> <log_folder>
randomnegotiator_dumbbot;<JAVA_ENV> -jar programs/RandomNegotiatorDumbBot-1.3.jar <ip> <port> <name> <nego_ip> <nego_port> <log_folder>
randomnegotiator_randombot;<JAVA_ENV> -jar programs/RandomNegotiatorRandomBot-1.3.jar <ip> <port> <name> <nego_ip> <nego_port> <log_folder>
human;<JAVA_ENV> -jar programs/ChatApp-1.0.jar <port> <nego_port> <ip> <nego_ip>
pulgarbot;<JAVA_ENV> -jar programs/PulgarBot-1.0.jar <ip> <port> <name> <nego_ip> <nego_port> <log_folder>
```

Figura 37. Archivo "availablePlayers.txt".

Añadimos la siguiente línea al final del archivo:

pulgarbot;<JAVA_ENV> -jar programs/PulgarBot-1.0.jar <ip> <port> <name> <nego_ip> <nego_port> <log_folder>

Guardamos el archivo y volvemos a la carpeta raíz del programa *gameManager*.

A continuación, copiamos el archivo del agente generado en el Anexo II de tipo *jar* en la carpeta "programs".

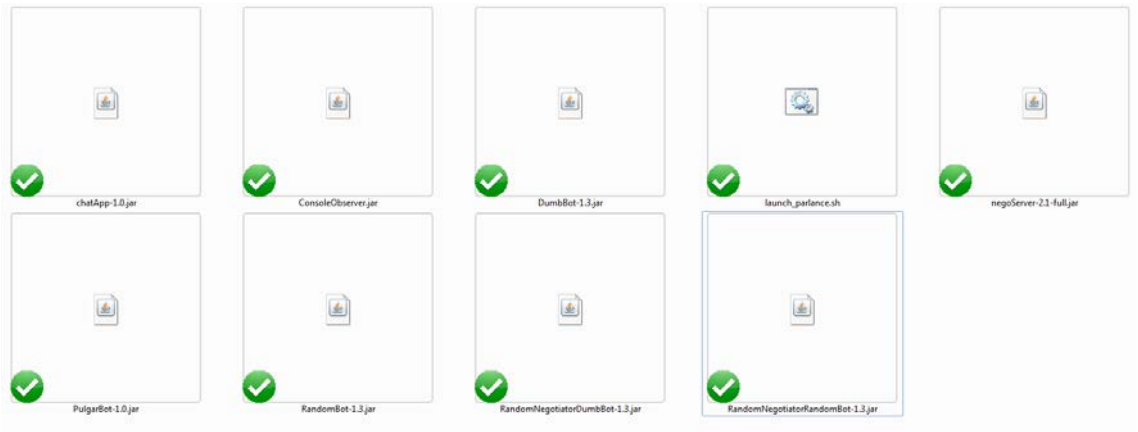


Figura 38. Carpeta "programs" con los agentes.

Una vez copiado, volvemos a la carpeta raíz para ejecutar *gameManager-1.0.jar*.

Si está asociada la extensión jar a Java se podrá ejecutar con doble clic, en otro caso habrá que usar la línea de comandos:

java –jar gameManager-1.0.jar

El programa nos recibe con la siguiente ventana:

Figura 39. Ventana inicial gameManager.

Haciendo clic en “dumbbot” se despliega la lista de agentes disponibles para ese jugador. En esa lista debe aparecer “pulgarbot”.

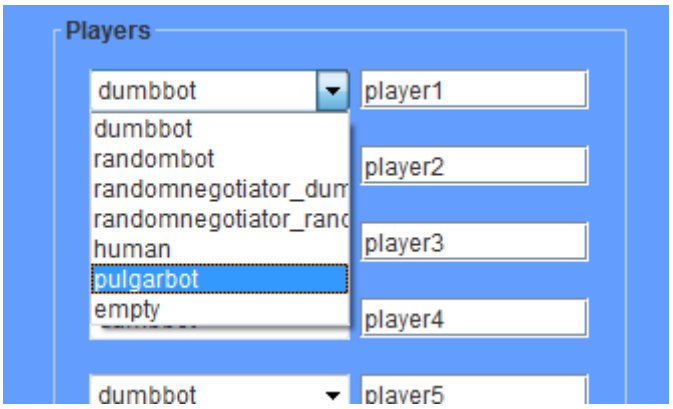


Figura 40. Selección del agente PulgarBot en DipGame.

Para una partida rápida, asignamos el agente “randomnegotiator_random” a todos los jugadores:

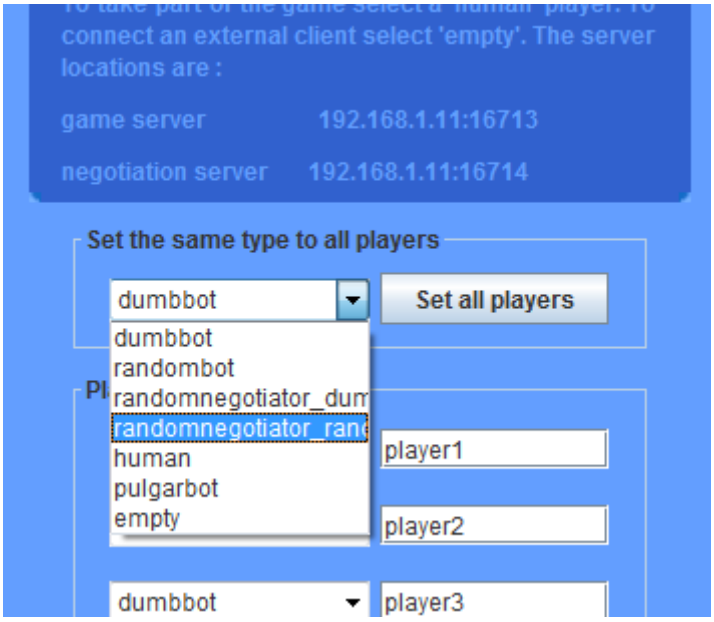


Figura 41. Asignación de "RandomNegotiatorRandom" a todos los jugadores.

A continuación se asigna a un jugador cualquiera el agente “pulgarbot”.

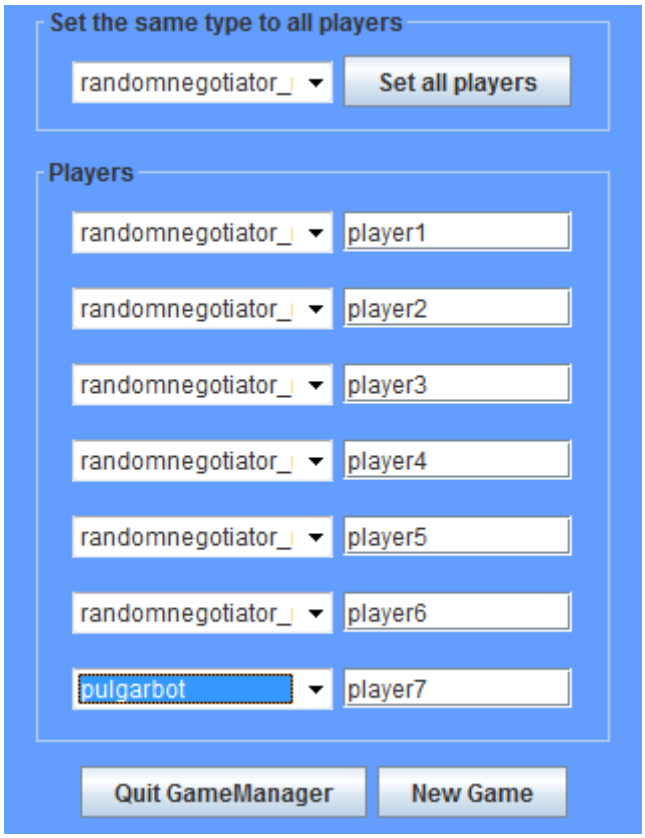


Figura 42. El agente "pulgarbot" competirá con los demás agentes.

Hacemos clic en "*New Game*" para iniciar la partida.

Se iniciará la ventana del registro del lenguaje DAIDE y la ventana del servidor DAIDE. La ventana del servidor DAIDE indica los jugadores, los agentes que usan, la potencia asignada al agente, el número de depósitos de suministros que controla y el número de unidades que tiene. También informa de la fase y año en el que se encuentra la partida.

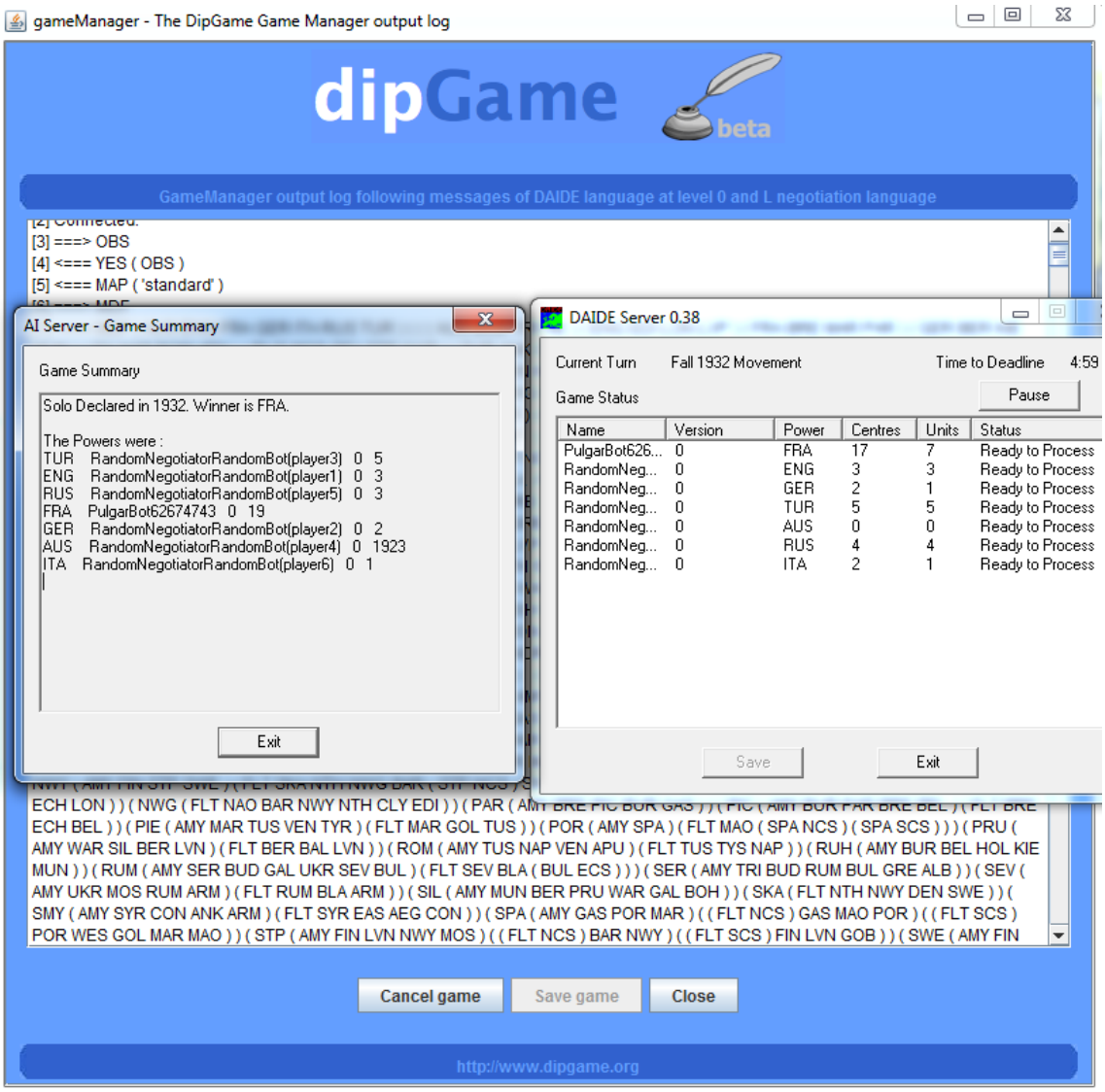


Figura 43. Partida finalizada de DipGame.

La partida termina con un mensaje del ganador y el estado de la partida.

Se generan varios ficheros de registro en la carpeta “logs”. El fichero relativo al agente siempre empieza por “dip_client_”.

Anexo IV: Gestión y planificación del proyecto

En este anexo se detallan la planificación y costes de la realización del proyecto.

IV.1 Planificación

El proyecto se ha distribuido en 9 tareas a lo largo de 11 meses.

Número	Tarea	Inicio	Fin	Horas
1	Documentación Agentes	1/11/13	15/01/13	175
2	Aprendizaje framework DipGame	16/01/14	28/02/14	140
3	Planificación de objetivos	3/03/14	10/03/14	25
4	Análisis del agente	11/03/14	1/04/14	155
5	Diseño del agente	2/04/14	12/05/14	195
6	Implementación del agente	19/05/14	26/07/14	220
7	Pruebas	28/07/14	5/08/14	40
8	Memoria	15/08/14	29/08/14	95
9	Revisión de la memoria	10/09/14	24/09/14	30

Tabla 18. Planificación de las tareas del proyecto.

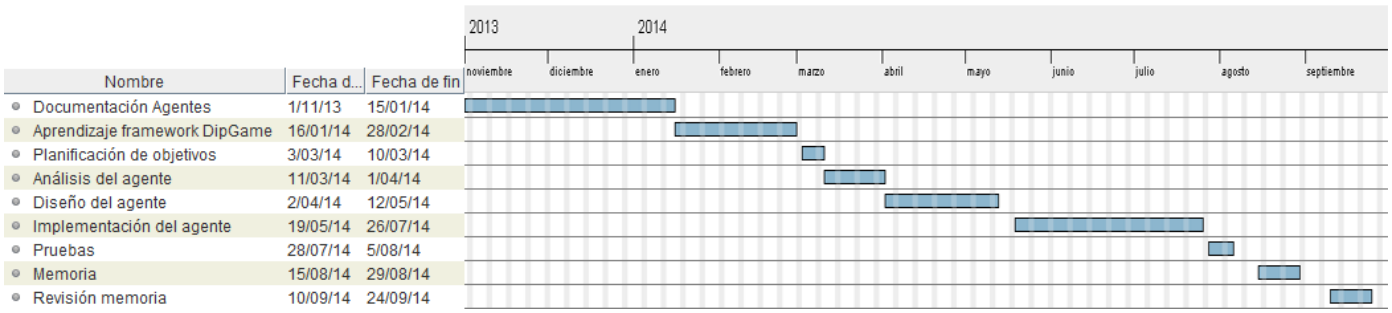


Figura 44. Diagrama de Gannt de la planificación del proyecto.

IV.2 Costes materiales

Para la realización del proyecto se ha requerido de un ordenador para el desarrollo del agente y una impresora para las revisiones de la memoria. La memoria USB permite el traslado del proyecto y como copia de seguridad del mismo.

Material	Cantidad	Precio	Total
Ordenador	1	1350 €	1350 €
Impresora Láser	1	110 €	110 €
Folios	1	3 €	3 €
Memoria USB	1	8 €	8 €
		Total	1471 €

Tabla 19. Materiales y costes asociados al proyecto.

IV.3 Costes personal

El personal requerido para el proyecto han sido de un jefe de proyecto que establezca los objetivos y realice la documentación del mismo, un analista que diseñe el agente y un programador junior que lo implemente y realice pruebas.

Posición	Cantidad	Tareas	Coste/h	Horas	Coste total
Jefe de proyecto	1	1, 2, 3, 8, 9	10,46 €	375	3922,5 €
Analista	1	2, 4, 5	10,26 €	400	4104 €
Programador junior	1	2, 6, 7	6,58 €	300	1974 €
			Total	**1075**	**10000,5 €**

Tabla 20. Coste del personal asignado al proyecto.

El coste por hora se ha determinado por el "Convenio colectivo estatal de empresas de consultoría y estudios de mercado y de la opinión pública" [26].

Tipo	Coste
Materiales	1471 €
Personal	10000,5 €
Total	11471,5 €

Tabla 21. Coste total del proyecto.

El coste total del proyecto asciende a la cuantía de ONCE MIL CUATROCIENTOS SETENTA Y UN EUROS CON CINCUENTA CÉNTIMOS DE EURO.

Fin de documento